Test Suite Optimization by Combining Modification-aware and Time-aware Selection using Integer Linear Programming

Florian Förg

Seminar: Software Quality
Advisor: Raphael Nömmer
Technical University of Munich
florian.foerg@tum.de

Abstract. Testing is an essential part of the software development lifecycle, as it verifies the quality of the software and to ensures that the resulting product stays maintainable and extensible. However, a long execution time of these tests can slow down the development process and may lead to a decrease in productivity. Furthermore, wasting ressouces leads to an increase in costs. Test Impact Analysis aims to reduce the execution time of test suites by using the techniques of test case selection and prioritization. The main idea is toidentify a subset of test cases that still produce significant results. This thesis introduces the idea of splitting up the selection process into one modification-aware and one time-aware selection step. By adopting this approach, it becomes possible to identify a subset of tests that exclusively cover changes and optimize the selected set with respect to different tested changed methods. The approach is evaluated with a dataset of real-world bugs provided by Defects4J. It turns out that in chances of detecting an error can increase by 2% when considering all prioritization techniques. However, with higher maximum execution times, we encounter some scalability issues, resulting in 1.8% less efficient results than when we use the original approach with the prioritization technique $ADDITIONAL\_COVERAGE\_PER\_TIME$.

Keywords: Test Impact Analysis · Test Case Selection · Integer Linear Programming

1 Introduction

A commonly used technique to increase the efficiency of a software development team and the quality of the produced code by early detection and resolution of integration issues, is to implement a Continuous Integration (CI) system. These systems promote the frequent integration of developers' local working copies into a remote repository. At regular intervals, such as daily or even with every commit, the CI system is activated to build the software and execute the test suite. If the build or any of the executed tests fail, the code can be reviewed and debugged to ensure the delivery of a functional product.

Motivation With growing test suites, the execution time of *retesting all* test cases can become a bottleneck [5]. The execution time of the test suite can slow down the development process and delay release of software. This leads to a decrease in productivity and a waste of resources.

Therefore, big test suites are often removed from the CI system and consequently are being run only rarely, which results in three major problems: (1) errors can hide and overlap each other, (2) the source of an error can potentially be in a bigger range of code, and (3) developers must understand in detail their code that was written some time ago [5].

To solve this problem three approaches have been studied: test minimization, test selection and test prioritization. Test Impact Analysis (TIA) which is the focus of this study, combines test selection and test prioritization.

TIA selects a prioritized subset of test cases that are affected by changes, reducing waiting times for test execution. The subset of impacted tests is computed by analyzing recent code changes, which are typically tracked by a Version Control System. The test cases that cover the changed lines of code are selected and prioritized. By prioritizing test cases, incorrect behavior can be detected earlier.

This study explores the concept of combining multiple selection techniques to identify an appropriate subset of tests that should be run within a given maximum runtime. In previous research, selection methods have focused on solely on modification-based selection[9] or time-aware selection [15]. The former does not consider the runtime of tests, while the latter fails to filter out tests that execute unchanged code. By introducing an Integer Linear Programming solution in combination with an existing modification-aware selection approach, both issues are addressed. Consequently, it becomes possible to achive an approximately 2% better chance to detect a failure in a scenario of different maximum runtimes and techniques to priotitize the tests as shown in the experimental evaluation in section 5.

The following sections provide an overview of the current state of Test Impact Analysis using the code analysis software Teamscale. Firstly, the concept of TIA is introduced. Next, we discuss various approaches for test case selection and prioritization. Subsequently, the theoretical and practical implementation of a two-step selection approach is explained, followed by an evaluation in the final section.

2 Definitions and Tools

This chapter lays out fundamental definitions that are useful for understanding the thesis in detail. These definitions become relevant when we formally define our problem and approach in section 4. A second part describes important software tools that we use.

2.1 Definitions

Test suite: A test suite $S = \{T[1], T[2], \ldots, T[|S|]\}$ is a set of test cases $T[x]$. $|S|$ is the number of tests that are elements of S. A project has exactly one (potentially empty) test suite that contains all test cases of the project. We assume that test suites are not empty and finite.

Test case: A test case T (also only "test") is an element of a test suite S. It is a sequence of actions that is executed by a test runner. A parametrized test, meaning a test case that allows multiple test scenarios to be executed with varying input values, is counted as a single test case.

- A test case can be *failing* if at least one assumption made within the test does not hold true. These assumptions are the expected conditions, inputs, or outputs that the test case is designed to validate.
- A test case is *new* if it was not executed before as part of CI.
- A test case is *modified* if it was executed before as part of CI, and the source code of the test case has changed since the last execution.
- A test case is *relevant* if it tests code that has been changed since the last execution, if it is new or modified.

We define the priority of a test case $P(T[x], PT)$ as a value that indicates how important it is to execute $T[x]$. The priority of a test case can differ with different prioritization techniques (PT). After calculating the order with a certain PT, test cases with a higher priority will be executed before test cases with a lower priority.

Relevant set: A relevant set $rS(S)$ is a subset of a test suite S. It contains all test cases $T[x] \in S$ that are *relevant*. The relevant set is computed by analyzing the changes of the code under test, which are tracked by a VCS. The relevant set can be *prioritized* by ordering test cases with respect to a certain prioritization technique. The prioritized relevant set of a test suite S with respect to a prioritiazion technique PT is called $prS(S, PT)$.

All failures/First failure: All failures describes the time that it needs to find all failing test cases of a test suite S. First failure is the time that it takes to find at least one failing test of a test suite S. Efficient test sets aim to mimimize all failure/first failure times (faster results).

Integer Linear Programming problem: An Integer Linear Programming (ILP) problem is a mathematical optimization problem in which both the objective function and the constraints are linear, and the decision variables are required to take on integer values. The objective function in an ILP problem represents the quantity that you want to maximize or minimize. The constraints in an ILP problem are linear inequalities or equations that restrict the values of the decision variables. Decision variables represent the quantities of interest that we want to determine or optimize. Solving an ILP problem involves finding the values of the decision variables that optimize the objective function while satisfying all the constraints. There are several well known algorithms solving these problems, for example the simplex algorithm [3].

2.2 Tools

For the implementation of the approach described in section 4.5 and the evaluation in section 5, we use the programming language Java. Furthermore, the following tools are important in the context of this study:

Teamscale Teamscale is a commercial software quality analysis tool developed by CQSE GmBH that facilitates the monitoring of software quality and offers immediate feedback. It connects to the source code repository and analyzes each commit. It supports uploading and analyzing external data, such as code coverage information.

Teamscale implements Test Impact Analysis as the underlying basis for this thesis. To implement our approach, we utilize the data obtained from the analysis to refine the selection process using ILP. However, no modifications are made to the existing modification-aware selection and prioritization techniques. The TIA as implemented in Teamscale is used to assess and compare the effectiveness of our modified approach. Teamscale incorporates various prioritization techniques that our evaluation depends on.

IBM CPLEX Optimizer CPLEX is a mathematical optimization software package. It provides a powerful set of tools and algorithms for solving linear programming , mixed-integer programming, quadratic programming, and convex quadratic programming problems. It also supports various specialized techniques, such as network optimization, stochastic programming, and constraint programming.
It is widely used in industry and academia to tackle complex optimization problems [8].

Most relevant for this thesis is that it provides an API for solving Integer Linear Programming problems as an external Java library [7] .

3 Related Work

The following section gives an overview of the previous research on TIA. It is divided into four parts. First, the general concept is discussed. A second part deals with the test selection problem. Afterwards, different approaches to solve the prioritization problem are presented. Finally, the current Teamscale status that is relevant for evaluating our concept, is introduced.

3.1 General Concept

Usually Test Impact Analysis is split up into two parts. The first part is the "Selection" of test cases. It solves the test selection problem as introduced in 2.1. We use one fixed selection technique.

The second part is the "prioritization" part which solves the test case prioritization problem by using a prioritization technique PT. After both parts are completed, we get a prioritized relevant set $prS(S, PT)$ [5].

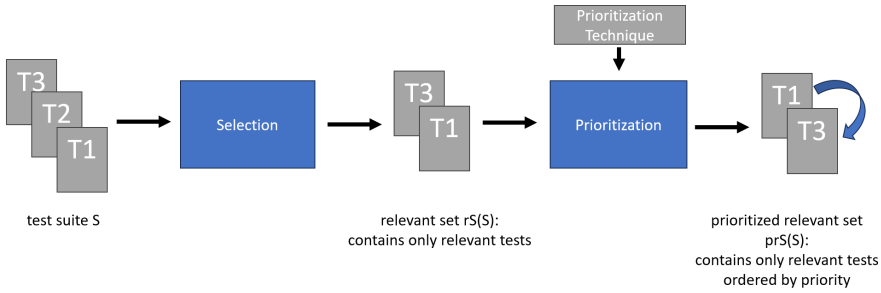

Fig. 1. Test Impact Analysis Concept

3.2 Test Case Selection

The following research assumes a setup as described above with (1) a basic code base and (2) the same code base after some changes were applied.

Test selection problem The test selection problem is the task to select a subset of test cases from a test suite S that maximizes the test suite's ability to detect faults in a software system while minimizing the cost and time required for test execution. The goal is to achieve adequate test coverage and fault detection capability while optimizing the resources involved in the testing process. Most test selection techniques are exclusively *modification-aware* (tests are considered depending on testing code changes) or *time-aware* (tests are considered depending on their runtime). As part of this thesis we will combine both approaches. The relevant set $rS(S)$ is a solution to the test selection problem.

Research on the topic of regression test selection provides far over 20 published techniques. In literature, they are classified in several ways:

(1) safe/unsafe: Safe techniques T guarantee that every defect found by the test suite S is also found by $rS(S)$ if the selection was made by using T. On the contrary, unsafe techniques do not have this characteristic [12]. In practice, implementing safe techniques can be challenging due to various factors such as timing constraints, configuration files, and other external dependencies that can lead to the failure of tests. These real-world complexities often hinder the effective application of safe techniques.

(2) Modification-aware selection techniques: Selection approaches are classified into three major topics which focus on modification-aware selection [2][1]:

The *Firewall Approach*, introduced by Leung and White [14], is based on the idea of dividing the code base into several modules, which are viewed as autonomous subsystems. In this context, a "module" usually refers to a discrete component or unit of a software program that is tested and developed independently, and during integration testing, these modules are combined to form the working program to emphasize interactions and interfaces between them. During the development process of a module M1, one of three things can happen: (1) no changes occur in M1, (2) M1's code is changed, and/or (3) M1's specifications are altered.

To select the relevant tests, a dependency graph is utilized to describe which modules call each other. Modules that are in state (2) and/or (3) are marked as changed and subsequently become part of the "firewall." Moreover, any modules that rely on services provided by the firewall modules are also included directly and transitively. Only the firewall-modules' tests are then executed. Example of techniques of this category: Rosenblum's and Weyuker's approach of a coverage information based RTS from 1997 [10].

DejaVu-based techniques use a more precise granularity to represent code segments than complete modules. These can be classes or even statements. The idea comes from Rothermel and Harrold as they invented their safe algorithm "SelectTests" in 1993 [11]. Most of these DejaVu techniques use control flow graphs for each individual entry to model dependencies. The goal is to find a minimum set of tests that are relevant for testing the changes. The techniques belonging to this category are usually slow and need a high computational effort. Therefore, there exist many optimized for certain problems versions, such as database-based applications. Also, the selection is very programming language dependent.

The last category is *based on specification/metadata*. It does not use source code or executable code but UML specifications or metadata in XML format for selecting relevant test cases.

(3) Time-aware selection techniques: In addition to the modification-aware techniques, there are time-aware techniques such as:

A *ILP-based* technique [3] as introduced by Lu Zhang et al. The goal is to find a optimal subset of tests given an additional time constraint. Therefore, the problem is formulated as a ILP problem. The approach introduced in this thesis adaptes the idea of using ILP for test case selection. The biggests difference to our approach is the realization of the objective function and the implementation. Furthermore, we will combine this approach with modification-based selection later on.

The categories can be combined as in this thesis. Many techniques are variants of others.

3.3 Test Case Prioritization

The **test prioritization problem** is the task of assigning a priority $P(T[x], PT)$ to each test case $T[x] \in rS(S)$. This helps to run tests that are most important and most likely to fail first. Consequently, you can get a more significant test result earlier which also allows to cut off test execution later on but having a result that is probably correct (better first failure). The prioritized relevant set $prS(S, PT)$ is a solution to the test prioritization problem.

Test prioritization techniques: Test prioritization techniques are used to optimize the testing process adjusting the order in which tests are executed. In the past, there have been different groups of techniques. The goal is to reach a maximum speed up in detecting the first/all failing tests of a test suite:

FIFOBaseline is most commonly used in practice. Tests are executed by the order they arrive in the code, and no ranking is being realized. It is the simplest but also the most ineffective technique [16]. The usage of this technique in practice is often not a deliberate choice but rather a result of its simplicity and potential limitations in understanding and implementing more advanced strategies.

Coverage-based prioritization tries to maximize the structural code coverage as early as possible. It can be realized in many different ways, for example as: line coverage, branch coverage, path coverage, function/method coverage, class coverage and so on [13].

Co-failing test prioritization is a result from past research [16] that has shown found out co-failing tests are a frequently occurring phenomenon. It describes the situation of two test cases $T[x]$ and $T[y]$ where $T[x]$ more than proportionally often fails if $T[y]$ fails and vice versa. They do not operate independently of each other. Data gained in previous test runs can be to predict the co-failing behavior. This can be used to dynamically re-prioritize test cases while running.

3.4 Selection and Prioritization in Teamscale

Teamscale incorporates a selection and prioritization mechanism to reduce the runtime of test suites. In this section, we will discuss the different techniques employed by Teamscale to achieve this.

Selection of test cases Teamscale employs a comprehensive test case selection process that involves an analysis phase to gather coverage information. This information can then be queried and utilized to perform the selection of test cases. The selection can be based on various criteria, including *covers changes*, *covers deleted, added or modified*, *previously failed*, *previously skipped* or *manually selected*. Importantly, this process operates at the method level, allowing for granular test selection. As part of this thesis we focus on test cases that are selected with *covers changes*. We will use the amount of covered changed methods to optimize the process.

Currently Teamscale only implements modification-aware selection.

Prioritization of test cases Teamscale provides several prioritization techniques:

- **NONE:** No prioritization is being executed.
- **FULLY_RANDOM:** This technique orders test cases in a completely random manner, without considering any specific criteria.

- **RANDOM_BUT_IMPACTED_FIRST:** Test cases are divided into two groups: (1) tests that are new or were modified, or tests that cover recent changes in the code, and (2) all other relevant tests. Within each group, the test cases are internally ordered randomly. However, the first group is executed before the second group.
- **ADDITIONAL_COVERAGE_PER_TIME:** Test cases are prioritized based on the coverage-to-time efficiency of methods that have not been covered yet.

Additionally, Teamscale allows users to specify a maximum runtime for the test execution. If the execution time exceeds the specified limit, Teamscale will omit lower-priority tests to ensure that the higher-priority tests are executed within the given time frame. The remaining time will then be allocated to executing the subsequent tests in order of their priority.

4 Approach

This section describes a modified approach for optimizing test selection in context of modification-aware and time-aware selection. It outlines the methodology to compute an optimized relevant set of test cases that maximizes the coverage of changed methods while adhering to specified time constraints. The section also discusses the utilization of Teamscale and an ILP solver for an implementation.

4.1 Initial Situation

For the following part, we assume that modification-based selection as well as test prioritization have already been implemented, as in Teamscale. Additionally, all test cases should run independent of each other, meaning that each test case can also be runned as the only test and would lead to the same result. Now, we want to introduce a second selection step that computes time-aware selection. The new concept is outlined as follows:

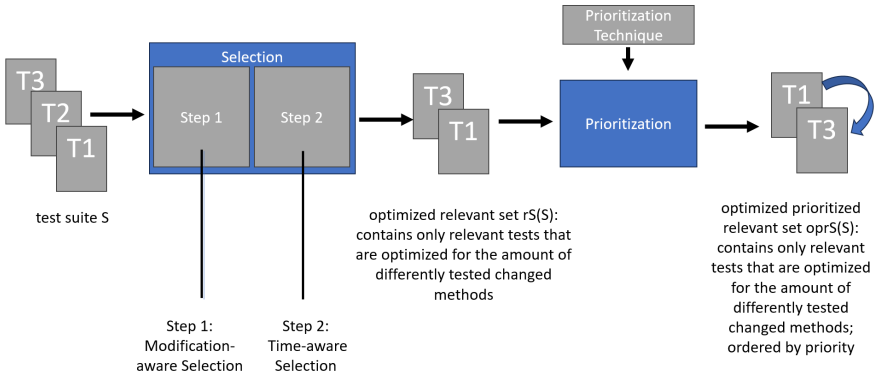

Fig. 2. Updated Test Impact Analysis Concept

4.2 Problem Definition

Given a test suite S as defined in section 2.1, the corresponding relevant set $rS(S) = T[1], ..., T[n]$ ($\implies n = |rS(S)|$) after the first selection step and a set M of methods that contains all recently changed methods, we aim to find an optimized relevant set $roS(S)$ that maximizes the unique tested methods $m \in M$ while ensuring it does not exceed the also given time limit t_{max}.

4.3 Solution Outline

To implement this approach, we will use Teamscale. It provides us with the functionality of the first subselection phase and the prioritization at the end. Furthermore, it determines the runtime of each test case, as well as the corresponding tested methods, which will be used to realize the second subselection phase.

To achieve optimization in the second phase, we combine the information from Teamscale and with an ILP solver implemented in CPLEX.

An ILP problem consists of constraints and an objective function. In our specific case, the problem statement includes the following constraints: 1) "the total execution time must not exceed the time limit t_{max}," and 2) "each test case can either be selected zero times or exactly once." Our objective function aims to maximize the number of unique changed methods tested by the selected set.

Our primary focus is on optimizing the testing of unique changed methods. As a result, during the second selection step, tests present in $rS(S)$ for reasons other than covering changes are highly likely to be excluded. This is because their inclusion does not contribute to maximizing the objective function. However, if there is additional time available, these tests may be included as random add-ons in the final results.

4.4 Optimization Problem

Formally, we define a function $f_{time} : S \to \mathbb{Q}$ that maps a test to the time it takes to run that case.

Additionally, we define a function $f_{coverage} : S \cup \{0\} \to C$ with $C \in P(S)$(power set) that maps a test to the set of statements it covers and 0 to the empty set $\emptyset$.

As described by Lu Zhang et al. in their approach for time-aware test-case priotization [15], we use desicion variables to decide whether a test case is selected for $roS(S)$. Goal of or approach is to find the decision variable for each test case.

$$x_i = \begin{cases} 1, \text{ if } T[i] \in roS(S) \\ 0, \text{ if not} \end{cases}$$

The product of a test case $T[i]$ with its corresponding decision variable x_i can be seen as:

$$x_i \cdot T[i] = \begin{cases} T[i], \text{ if } x_i = 1 \\ 0, \text{ else} \end{cases}$$

With $rS(S == T[1], ..., T[n]$ and $n = |rS(S)|$ as stated in the problem statement, we abstract the problem of finding the optimal test suite to the following optimization problem, with x_i unknown:

First constraint that must hold is, that x_i can only be 0 or 1 because the test case can be selected or not, but nothing in between. In ILP this is called binary constraint. Formally, it is represented by:

$$x_i \in \{0, 1\} \tag{1}$$

Furthermore we know, that the summed up time of all test cases selected has to be less or equal with the maximum time. So we add the time constraint:

$$\Sigma_{i=1}^{n}(x_i \cdot f_{time}(T[i])) \leq t_{max} \tag{2}$$

Finally, we want to maximize the unique covered methods. Therefore, we define the following objective function which maximizes the cardinality of the set containing all methods that are tested by the selected subset of tests:

$$maximize : |\bigcup_{i=1}^{n} f_{coverage}(x_i \cdot T[i])| \tag{3}$$

To utilize Integer Linear Programming for solving the problem, we need to linearize the objective function. For this purpose, we create a summand for each method that is tested by at least one test in $roS(S)$.

We define a function $max(N)$ which represents the largest integer value of a set N. Then each max summand itself represents the maximum value among a set of all decision variables of tests that cover the corresponding method. Consequently, if a method is covered by at least one test, the summand becomes one, resulting in the maximization of the objective function. This enables us to find an optimal solution that maximizes the number of unique tested methods.

In cases where multiple selected test cases cover the same method, the value of the summand remains one, as $max(\{1, 1\}) = 1$. This aspect reflects the uniqueness requirement stated in the problem statement. This linearized function has the following form:

$$maximize : \Sigma_{m \in M} max(\{x_i | m \in f_{coverage}(T[i])\}) \tag{4}$$

Note that there may not always be a unique solution. For instance, different subsets of test cases could cover the same maximum number of unique methods and still run within the given time constraints. In our approach, we can guarantee finding one of the optimal solutions, but it is important to acknowledge the possibility of multiple equally optimal solutions.

Also note that solving the ILP problem is NP-hard. Consequently, even when always at least one optimal solution exists, in pratice we can often only approximate a result that is as optimal as possible.

4.5 Implementation

This part contains a simplified description of how the second selection step described above is realized with Java, CPLEX and Teamscale. The following code segments are written in pseudo code with Java syntax:

To select an optimized subset of tests, we require the relevant tests obtained from the first selection step, the methods that need to be tested, and the maximum runtime. This necessary information is provided by Teamscale.

```
1 public class ILPSelection
2     public static <T extends TestCase> Collection<T> selectTests(
3     Collection<T> relevantTests, Set<MethodId> methodsToTest,
4     @NotNull Long maxTestSuiteRuntime)
5     {
6         ...
```

In the next step, we instantiate an $IloCplex$ object, utilizing the functionality provided by CPLEX, to represent our Integer Linear Programming problem. Furthermore, we define an array called x of type $IloIntVar$, which represents binary (0 and 1) values. Each element $x[i]$ in our code corresponds to x_i as described in Section 4.4. By initializing the array as $boolVarArray$, we imply the constraint from equation (1).

```
    ...
    IloCplex cplex = new IloCplex();
    IloIntVar[] x = cplex.boolVarArray(relevantTests.size());
    ...
```

With the setup complete, we can now define our time constraint function. To accomplish this, we multiply the runtime of each test by the corresponding unknown boolean variable we are seeking and sum up all the values, as described in equation (2).

```
    ...
    //Create time constraint
    IloLinearNumExpr timeConstraint = cplex.linearNumExpr();

    for (TestCase test : relevantTests) {
        // Add the decision variable multiplied by the runtime of the test
        timeConstraint.addTerm(test.getDuration() * x[i]);
    }
    ...
```

In addition, we set up the objective function to maximize the number of unique tested methods. For each method m in $methodsToTest$, we create a summand that represents the maximum value among all x_i corresponding to tests covering method m. This corresponds to equation (4). As a result of this step, the initialization runtime is $O(|relevantTests| \cdot |methodsToTest|)$.

```
    ...
    //Create objective function
    IloIntExpr objective = cplex.intExpr();

    for (MethodId id : methodsToTest) {
        List<IloIntVar> variablesToMaxExpression = new ArrayList<>();

        for (TestCase test : relevantTests) {
              //If test covers method --> add to list of variables considered for max
                   function
             if (test.getCoveredMethods().contains(id)) {
                testsCoverMethod.add(x[i]);
             }
          }

        //If method is covered by at least one test, add max summand to objective
            function
        if (testsCoverMethodArray.length > 0) {
            objective.add(cplex.max(variablesToMaxExpression));
        }
    }
    ...
```

Finally, the constraints and objective function need to be associated with the ILP object. Subsequently, CPLEX can solve the problem and return the required x_i values in the array x. As a final step, we create a new result list that includes all tests from $relevantTests$ whose corresponding $x[i]$ value is 1.

```
    ...
    //Set objective function to be maximized
    cplex.addMaximize(objective);

    //Set runtime to be <= max runtime
    cplex.addLessEquals(timeConstraint, maxTestSuiteRuntime);

```

```
47      cplex.solve();
48
49      List<T> result = new ArrayList<>();
50
51      //Add selected tests to result
52      for (int i = 0; i < releventTests.size(); i++) {
53          //Desicion variable must be 1 to be selected
54          if (cplex.getValue(x[i]) == 1) {
55              result.add(relevantTests.get(i));
56          }
57      }
58
59      return result;
60  }
```

5 Evaluation

We experimentally evaluated the effectiveness and efficiency of the our approach of selection optimization with an additional selection step using Java bugs from Defects4J database. Our experiments address the following research questions:

- *RQ1*: How does the performance of our approach compare to other state-of-the-art Teamscale techniques in terms of fault detection effectiveness? Are there scenarios where our approach outperforms other techniques, and vice versa?
- *RQ2*: How does our approach performe in terms of first failure & all failure finding times? Does it influence how fast we get a valuabel result?

5.1 Setup and Methodology

Defects4J is an open-source benchmark suite specifically designed for evaluating the effectiveness of automated software testing techniques, particularly in the field of program repair and debugging. The dataset contains a wide variety of real-world Java bugs (and corresponding fixes) and a comprehensive test suite for each program version, consisting of a combination of passing and failing test cases [6]. Defects4J covers a wide range of scenarios. We will use version 2.0.0 for evaluating the introduced approach with all bugs from the projects jsoup, commons-csv and commons-math.

We set up Defects4J on WSL2.0 running Ubuntu 22.04 on a Windows 10 computer. It uses an AMD 2600x CPU and 16GB 3600MHz RAM. A local developer version of Teamscale 8.9 is running on the Windows OS as well. Defects4J requires Apache Maven (version 3.6.3 with Java 1.8), Perl 5 (version 34), SVN (version 1.14.1), and Git (version 2.34.1).

In the following, we will use the *TestExecutionListener*[1] to track tests that are being executed by JUnit4 and Maven Surefire. This information will be used by the *Teamscale-JaCoCo-Agent*[2] to upload coverage information to Teamscale.

Now, we set up a project in Teamscale that we want to evaluate, with the same name as it has in Defects4J.

For the following evaluation steps, we use scripts[3] that automate the process:

[1] https://github.com/Raphael-N/TestwiseExecutionListener
[2] https://github.com/cqse/teamscale-jacoco-agent
[3] https://github.com/florianfoerg/test-impact-analysis-thesis/tree/main/Setup/data-collection

1. In the first step, we modify the Defects4J project configurations so that the coverage information will be uploaded.
2. Then, we iterate through every active bug of a project provided by a list of Defects4J and run the tests for the buggy and fixed versions.This triggers a upload up the coverage information to Teamscale.
3. After the upload is finished, we automatically reanalyze the Teamscale project.
4. Now, we can send a request to get the prioritized relevant set for each bug we uploaded the information for before. The results will be stored in a *.csv* file. It contains meta information for each data set and additionally the columns:

ILP Used	*Prioritized (Optimized) Relevant Set*	*Failing Tests*

ILP used is a boolean value that describes if the result was found with or without the second selection step.

Prioritized (optimized) relevant set is the ordered list we find with Teamscale after TIA was executed. When we use the additional ILP step, it is optimized.

Failing tests is a list of tests failing due to the bug. We get this value from Defects4J.

We test thirteen different combinations of *prioritization technique* PT and *maximum execution time* t_{max} in milliseconds. First, we request the data for $PT = NONE$ and $t_{max} = null$. This way, we get the data after the first selection step, which we can later use to verify that the upload worked correctly and no errors occurred. Then, we gather data for all combinations of $PT \in \{\ NONE,\ FULLY\_RANDOM,\ ADDITIONAL\_COVERAGE\_PER\_TIME\ \}$ and $t_{max} \in \{20\%, 40\%, 60\%, 80\%\}$ in percent of the total relevant set runtime. Other parameters used by Teamscale are set to baseline = time_stamp_buggy_commit, end = time_stamp_fixed_commit, include-non-impacted = false.

We run the process described above first for a Teamscale version that does not implement the second selection step, and afterwards for one that does. If not defined differently, we use all data values collected to get the resulting statistics.

It is important to mention that we modify our ILP selection implementation in a way that we set a time limit of 20 seconds to solve the ILP problem for CPLEX. To realize this, we set a time limit to our *IloCplex* object in the implementation. Consequently, we may not find an optimal solution, as described in section 4.4. We gurantee that the result is feasible and as close to the optimum as possible. Altought CPLEX is highly optimized, ILP problems are NP-hard, and finding a verified optimal solution would take too much time for a large number of test cases and methods that need to be tested. Moreover, it could only be solved in exponential time. Setting a time limit helps strike a balance between computational resources and the desired level of coverage. If the time limit is too short, the ILP algorithm may not have sufficient time to explore a significant portion of the solution space, potentially far away from the optimal solution. We set up CPLEX to find a balance between optimality and feasibility of solutions, providing fast but good results.

Furthermore, note that the *prioritized (optimized) relevant set* is computed for the changes between the buggy commit and the fixed commit, not the other way around. This is valid since we only want to check which tests are selected. Potentially, the buggy version could represent our intended behavior that was changed.

In the last step, we can use visualization scripts[4] to evaluate the collected data.

[4] https://github.com/florianfoerg/test-impact-analysis-thesis/tree/main/Setup/data-visualization

5.2 Results and Discussion

RQ1: How does the performance of our approach compare to other state-of-the-art Teamscale techniques in terms of fault detection effectiveness? Are there scenarios where our approach outperforms other techniques, and vice versa?

We first examine the likelihood of the test selected by TIA to uncover at least one failure. Additionally, we explore the changes in results achieved through the second selection step when considering all collected data values. It is important to note that these changes differ when solely using the data values collected with the *ADDITIONAL_COVERAGE_PER_TIME* prioritization technique.

Fig. 3. Comparison of failure finding rates between TIA with additional ILP selection step and without. All data values with all *PT*s considered.

Fig. 4. Comparison of failure finding rates between TIA with additional ILP selection step and without. Only data values considered that were found with $PT = ADDITIONAL\_COVERAGE\_PER\_TIME$.

First, we can see that the failure detection rates for the ILP solution stay the same when we use all *PTs* like in Fig. 3 or only restrict us to one like in Fig. 4. This is achieved by satisfying the time constraint before ranking the tests, eliminating a need for post-ranking test cutoffs. As a result, the *PT* does not influence the failure detection rates when using the two-step implementation. We can observe that this behaviour differs from the original TIA. Here the likelihood of detecting a failure is influenced by the prioritization technique. This is primarily due to the test cutoff process employed in Teamscale, which involves removing tests based on their rank (as described in section 3.4).

When only considering the data collected with the *ADDITIONAL_COVERAGE_PER_TIME* prioritization technique, we observe a improvement in the performance of the non-ILP method. This comes as the most time-inefficient tests are know ranked last and consequently will be cut off. The most time-efficient tests will stay in the result. It is important to note that while our approach aims to optimize the selected test set, it is not always superior to the alternative methods. This is the result of two reasons:

1. Cases in which multiple tests may cover the same methods, but potentially only one of them successfully detects a faulty branch. In such cases, if the test that identifies the broken branch does not maximize the overall test coverage, our approach would not select it. Consequently, there may be instances where the alternative method, despite its different selection process, outperforms our approach in terms of detecting specific types of faults.

2. Non optimal solutions that occur as a result of the time limit in ILP solving. We can see that the higher the maximum execution times become (especially 60% and 80%), the less improvements to the next higher maximum execution time can be seen. This is, as the time constraint excludes less possible results which makes it more difficult to find a optimal solution in a appropiate time.

In summary, the new approach achieves an failure detection rate of approximately 60.1%. Comparatively, when not using ILP and without all prioritization techniques, the failure detection rate is approximately 58.1%. This indicates that our approach is 2% more likely to detect failures than the non-ILP method without, using all prioritization technique. When not using ILP and employing *ADDITIONAL_COVERAGE_PER_TIME* as the prioritization technique, the failure detection rate is approximately 61.9%. The non-ILP method now performs 1.8% than our approach.

RQ2: How does our approach performe in terms of first failure & all failure finding times?

For the following diagram we define P1 as placeholder for priotization technique *NONE*, P2 for *FULLY_RANDOM* and P3 for *ADDITIONAL_COVERAGE_PER_TIME*. For *RQ2*, tests that do not identify any first failure or fail to detect all failures are excluded from the corresponding statistics. All other data values collected, like described in section 5.1, are considered.

Fig. 5. Data points for first failure detection times ordered by priotization technique and usage of ILP. Each point represents one tested bug.

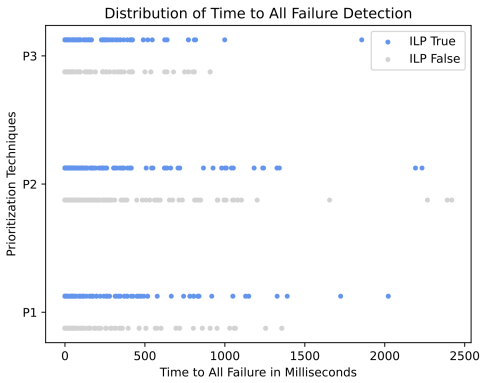

Fig. 6. Data points for all failures detection times ordered by priotization technique and usage of ILP. Each point represents one tested bug.

As observed in the graphics, the first failure and all failures detection times are significantly influenced by the prioritization technique used. It is evident that the majority of bugs fail within the first 100 ms in all cases.

Of particular interest is the comparison between the distribution of detection times with and without the use of ILP, specifically for the *ADDITIONAL_COVERAGE_PER_TIME* (P3) technique. In this case, the distribution of detection times is quite similar for both scenarios. This can be attributed to the fact that the selected test cases exhibit high coverage within a short duration, and it is likely that these tests are present in both sets after the selection. Since we also employ the P3 technique after ILP selection, the order of the selected test cases remains the same. The median times for first failure detection are 18 ms with ILP and 20 ms without ILP, while the median times for all failures detection are 68 ms with ILP and 80 ms without

ILP. Consequently, the similarity in test case selection and order explains the resemblance in the distribution of detection times.

When using the *FULLY_RANDOM* (P2) technique for prioritization, the differences in distribution are more pronounced. This is because the selected test sets differ randomly, and the execution order is also randomly assigned with or without ILP. As a result, the median time for first failure detection is 58 ms with ILP and 99 ms without ILP, while the median time for all failures detection is 155 ms with ILP and 301 ms without ILP. The substantial differences can be attributed to the random nature of test case selection and execution order.

The most significant differences are observed when no prioritization technique (*NONE* - P1) is employed. Without a prioritization technique, tests that cover the same methods are likely to run close to each other in terms of execution time. When we apply the selection without second step and truncate tests at the end, the preceding tests are likely to cover the same methods. This results in a significantly different test set compared to the selection with ILP, as the latter aims to maximize the coverage of different methods. Consequently, the median times for first failure detection are 48 ms with ILP and 93 ms without ILP, while the median times for all failures detection are 156 ms with ILP and 340 ms without ILP.

Overall the performance with the second ILP selection step is better in all categories. By selecting a diverse set of test cases ILP eliminates redundancy and focuses on maximizing the coverage of unique methods. This reduction in redundancy allows for more efficient testing, as valuable resources are not wasted on repetitive tests that provide little additional information. This comprehensive coverage ensures that a wider variety of program behaviors and interactions are tested within a short time, leading to faster failure detection times. As before, the influence of the second selection step can also be minimized by selecting P3 as priotization technique.

5.3 Threats to Validity

While selecting an optimal subset of test cases based on unique covered methods is a valuable approach, there are certain threats to the validity of the results. It is important to acknowledge and consider these potential limitations:

1. **Implementation:** In our experimental study, the primary concern regarding internal validity is the potential presence of errors or flaws in the implementation of the techniques we investigated. To mitigate this threat, we took several measures. Firstly, we incorporated reliable and well-tested components, such as IBM Cplex and Teamscale, which have demonstrated robustness in previous applications [4]. Additionally, prior to the execution we implemented unit tests to ensure the code's functionality.
2. **Multiple Solutions:** The process of selecting an optimal subset may yield multiple solutions that fulfill the criteria of maximized unique covered methods. However, these solutions may not be equivalent in terms of error finding capabilities. Different test cases can target the same method but cover different branches or scenarios within that method. Consequently, evaluating the same project multiple times may lead to slightly different results due to the varying combinations of test cases. To address this issue, we collected a large number of data values (2580 from 215 bugs) to ensure a robust and representative sample, enabling us to calculate a reliable mean value for evaluating the performance of the approach across various bug instances.
3. **Time Limit:** Another important factor to consider is the time limit set for the execution of the techniques. Since ILP is known to be NP-hard we had to set a time limit for testing. As a consequence, the performance of the computer can significantly impact the quality of the results. The more powerful the CPU, the more optimal solutions the ILP algorithm can explore, potentially leading to better results in terms of unique covered methods. However,

due to practical constraints, it may be necessary to set a time limit for the execution in real-world scenarios.

4. **Representativeness:** Threats to external validity revolve around the extent to which the results of our experiments can be generalized. As we have seen, the more complex the ILP problem becomes, the less optimal solutions we find. We run into some scalability issues here. When using the approach with projects that have another amount of tests, bigger commits/commits over a longer time or another time limit, we might find other results. To mitigate this threat, we opted to evaluate our approach using Defects4J, a dataset comprising numerous real-world bugs with projects of different sizes (< 50 test cases to > 5000 test cases). By incorporating this diverse and extensive collection of bugs, we aimed to enhance the generalizability of our findings to a broader range of software development scenarios.

6 Conclusion

After an introduction (section 1) on Test Impact Analysis, we introduced a two-step approach (section 4) for test case selection using Integer Linear Programming. It aims to maximize the number of different tested methods that were changed for a given time constraint. In an experimental evaluation (section 5) we showed that it is about than 2% more likely to detect an error for different maxmimum execution times and priotization techniques. However, it is important to note that ILP is NP-hard and may encounter scalability challenges. Therefore, we had to accept the best solution to be found within a time limit of 20 seconds. As a result, the approach is about 1.8% less effective than when using the original Teamscale TIA process with the prioritization technique $ADDITIONAL\_COVERAGE\_PER\_TIME$.
For future work, it will be interesting to research how far away the found solutions are from the optimum and how the approach behaves within other environments (other hardware Teamscale is executed on, other time limits for the ILP solving process, other projects).

References

1. Dreier, F.: Obtaining coverage per test case. Ph.D. thesis, MA thesis. Munich, Germany: Technische Universität München (2017), https://www.cqse.eu/fileadmin/content/news/publications/2017-obtaining-coverage-per-test-case.pdf
2. Engström, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques. Information and Software Technology **52**(1), 14–30 (2010), https://www.sciencedirect.com/science/article/pii/S0950584909001219
3. Graver, J.E.: On the foundations of linear and integer linear programming i. Mathematical Programming **9** (1975), https://link.springer.com/article/10.1007/BF01681344
4. Heinemann, L., Hummel, B., Steidl, D.: Teamscale: Software quality control in real-time. In: Companion Proceedings of the 36th International Conference on Software Engineering. p. 592–595. ICSE Companion 2014, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2591062.2591068
5. Jürgens, E., Pagano, D., Göb, A.: Test impact analysis: Detecting errors early despite large, long-running test suites. Whitepaper, CQSE GmbH (2018), \brl{https://www.cqse.eu/fileadmin/content/news/publications/2018-test-impact-analysis-detecting-errors-early-despite-large-long-running-test-suites.pdf}
6. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. p. 437–440. ISSTA 2014, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2610384.2628055
7. Manual, C.U.: Ibm ilog cplex optimization studio. Version **12**(1987-2018), 1 (1987), https://perso.ensta-paris.fr/~diam/ro/online/cplex/cplex1271_pdfs/usrcplex.pdf

8. Nickel, S., Steinhardt, C., Schlenker, H., Burkart, W.: Decision Optimization with IBM ILOG CPLEX Optimization Studio: A Hands-On Introduction to Modeling with the Optimization Programming Language (OPL). Springer Nature (2022), https://link.springer.com/book/10.1007/978-3-662-65481-1
9. R, B., S, S.: Code coverage based test case selection and prioritization. International Journal of Software Engineering Applications **4**(6), 39–49 (nov 2013), https://doi.org/10.5121%2Fijsea.2013.4604
10. Rosenblum, D., Weyuker, E.: Using coverage information to predict the cost-effectiveness of regression testing strategies. IEEE Transactions on Software Engineering **23**(3), 146–156 (1997), https://ieeexplore.ieee.org/document/585502
11. Rothermel, G., Harrold, M.: A safe, efficient algorithm for regression test selection. In: 1993 Conference on Software Maintenance. pp. 358–367 (1993), https://doi.org/10.1109/ICSM.1993.366926
12. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. **6**(2), 173–210 (apr 1997), https://doi.org/10.1145/248233.248262
13. Yang, Q., Li, J.J., Weiss, D.: A survey of coverage based testing tools. In: Proceedings of the 2006 International Workshop on Automation of Software Test. p. 99–103. AST '06, Association for Computing Machinery, New York, NY, USA (2006), https://doi.org/10.1145/1138929.1138949
14. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability **22**(2), 67–120 (2012), https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.430
15. Zhang, L., Hou, S.S., Guo, C., Xie, T., Mei, H.: Time-aware test-case prioritization using integer linear programming. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. p. 213–224. ISSTA '09, Association for Computing Machinery, New York, NY, USA (2009), https://doi.org/10.1145/1572272.1572297
16. Zhu, Y., Shihab, E., Rigby, P.C.: Test re-prioritization in continuous testing environments. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 69–79 (2018), https://doi.org/10.1145/1572272.1572297