TU WIEN Informatics

Balancing Time and Test Coverage: A Time-Constrained Approach to Software Testing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

René Zumtobel, BSc
Matrikelnummer 11777705

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Thomas Grechenig

Wien, 28. August 2024

Unterschrift Verfasser | Unterschrift Betreuung

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Balancing Time and Test Coverage: A Time-Constrained Approach to Software Testing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

René Zumtobel, BSc
Registration Number 11777705

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

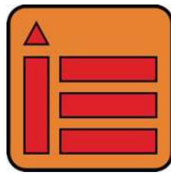
Vienna, 28th August, 2024

Signature Author

Signature Advisor

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



Balancing Time and Test Coverage: A Time-Constrained Approach to Software Testing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

René Zumtobel, BSc
Matrikelnummer 11777705

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 28. August 2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

René Zumtobel, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. August 2024

René Zumtobel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Automatisierte Tests, welche in einer Continuous Integration (CI) Umgebung ausgeführt werden, sind ein fester Bestandteil moderner Softwareentwicklung. Sie werden sowohl von Open-Source-Projekten als auch von Branchenriesen wie Google, Amazon und Facebook genutzt, um Fehler frühzeitig zu erkennen. Die Herausforderung besteht darin, dass mit zunehmender Größe eines Projekts und mit mehr geschriebenem Code die Laufzeit jeder einzelnen Pipeline steigt. Zentral ist die Frage, wie kann das Problem langer Pipelines gelöst werden? Die wohl einfachste Möglichkeit besteht darin, Ressourcen in horizontale und vertikale Skalierung zu investieren. Doch welche Alternativen gibt es?

Forschung, die sich mit diesem Thema befasst, konzentriert sich hauptsächlich auf die Auswahl von Testfällen und die Minimierung von Testsuiten. Eine Frage, die jedoch nur selten Beachtung findet: Kann die Testausführung auf ein bestimmtes Zeitbudget reduziert werden? Wie würde ein solcher zeitlich begrenzter Ansatz für Softwaretests aussehen? Welche Informationen müssen genutzt und welche Metriken optimiert werden? Wie effizient ist ein solcher Ansatz im Vergleich zur Durchführung aller Tests? Wie groß muss ein solches Zeitbudget mindestens sein?

Diese Arbeit soll diese Fragen beantworten. Einleitend wird ein Überblick über die Forschung zu diesem Thema gegeben. Anschließend wird eine Methode für einen solchen zeitlich begrenzten Softwaretestansatz unter Verwendung der Design Cycle Methode entwickelt. Ein Fallstudienprojekt aus der Industrie wird verwendet, um die Effizienz des Ansatzes zu evaluieren.

In über 40.000 Minuten Testzeit übertraf die entwickelte Methode die Ergebnisse des Kontrollansatzes und zeigte Ergebnisse, die mit denen einer vollständigen Pipeline vergleichbar waren.

Keywords: *Test Auswahl, Continous Integration, Automatisiertes Testing*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Automated tests executed within a Continuous Integration (CI) environment are a staple of state-of-the-art software engineering. They are utilized by everyone from open-source projects to industry giants like Google, Amazon, and Facebook. However, one major challenge remains: the bigger a project's scope gets, the more code is written, and the more tests accumulate, the higher the overall execution time. The issue of execution time can be addressed in numerous ways. An easy one is to allocate ever-increasing resources by scaling horizontally and vertically. What are the alternatives?

The research addressing this challenge centers primarily on test case selection and test suite minimization. One idea is rarely covered: Can a test suite be reduced to a strict time budget? What would such a time-constraint approach to software testing look like? What information needs to be utilized? How does such an approach compare to executing all tests? What time budget is sufficient?

This thesis aims to address these questions. It will start with an overview of the limited research on the topic. Afterwards, a method for such a time-constraint software testing approach will be developed using the design cycle method. An industry case study project will be utilized to evaluate the approach.

In over 40.000 minutes of testing, the developed method outperformed random testing and repeatedly showed results comparable to running a full test pipeline.

Keywords: *Test Case Selection, Continuous Integration, Test Execution*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Motivation . . . 2
1.3 Expected Results and Research Questions . . . 3
1.4 Structure . . . 4

2 Fundamentals **7**
2.1 Testing . . . 7
2.2 Continuous Integration/ Continuous Delivery . . . 10
2.3 Optimization Problems . . . 11
2.4 Evolutionary Algorithms . . . 12

3 Related Work **17**
3.1 Coverage Goals . . . 17
3.2 Test Case Selection and Prioritization . . . 19
3.3 Test Case Selection With Time Budget . . . 30
3.4 Test Case Generation . . . 31

4 Methodology **35**
4.1 Research . . . 35
4.2 Implementation . . . 36
4.3 Case Study and Analysis . . . 37
4.4 Research Questions . . . 37

5 Method Design **39**
5.1 Iteration Overview . . . 40
5.2 0. Iteration - Requirement Analysis . . . 40
5.3 1. Iteration . . . 41

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.4 2. Iteration . . . 46
5.5 3. Iteration . . . 55
5.6 4. Iteration . . . 61
5.7 5. Iteration . . . 63
5.8 6. Iteration . . . 68
5.9 7. Iteration . . . 77
5.10 8. Iteration . . . 80
5.11 9. Iteration . . . 83

6 Evaluation and Results **85**
6.1 Benchmark Validation . . . 85
6.2 Commit Size Analysis . . . 89
6.3 Ideal Time Budget . . . 93

7 Discussion **95**
7.1 Research Question 1 . . . 95
7.2 Research Question 2 . . . 96
7.3 Research Question 3 . . . 98
7.4 Research Question 4 . . . 99

8 Conclusion **101**
8.1 Threats to Validity . . . 101
8.2 Future Work . . . 102

List of Figures **103**

List of Tables **105**

List of Algorithms **107**

Acronyms **109**

Bibliography **111**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

Software Engineering is a field where issues are unavoidable. The longer a project prolongs, eventually, some will appear, and no matter the best intention of teams to keep problems under control, they can still be overwhelmed by them [23]. This is also true for bugs introduced during the development process. It is essential to spot them before they are deployed, as they can cause issues for users. A way to ensure quality and adherence to specifications is to employ Continuous Integration (CI). By maintaining a high test coverage and quality of tests, bugs that affect functionality due to side effects can be caught. If tests are executed within a CI environment, which triggers a pipeline that runs tests after every commit, then errors should be caught right after the commit.

This testing and the need for swift feedback comes at a high computational cost. Computational costs increase the bigger a project gets, as more functionality needs testing.

1.1 Problem Statement

Rogers [69] writes in his article on Extreme Programming, of which CI is part of the method set, that adding new features, as well as a growing code base, leads to increased build time and an increase of time required for the testing stage. This is caused by the growth in the number of tests. Rogers claims that runs of 30 minutes are not unusual and that the resulting wait time decreases productivity. This can result in fewer commits, which can cause more conflicts.

The increase in per pipeline runtime leads to three problems:

- Longer Time to Feedback: The wait time for developers until their pipeline is completed increases. The most significant problem is that this increase can be longer than the per pipeline increase if other pipelines are prior in the queue. This

issue can be mitigated by adding more runners and increasing parallelization, but at the cost of buying and powering more runners.

- Higher Computational Demands: As each pipeline lasts longer, it issues more instructions to the CPU, leading to increased computational needs.
- Higher Power Consumption: If no other changes are made, an increase in computation leads to an increase in power consumption.

To address these issues, this thesis has the goal of developing a method to reduce a test suite to a specific time budget. This method utilizes the information from the Source Code Management (SCM) to determine which tests relate to code changes. The assumption is made that the test suite ran successfully with the last commit. Therefore if there are new errors, they have to be covered by the tests related to changed code areas or the new tests introduced in the commit. If this is not the case, then the assumption is that they are not covered at all.

This work addresses multiple questions to fit these tests into a time budget: Which tests are associated with changes introduced in the new set of commits? What tests must be selected to get the highest probability of finding introduced bugs? How can this set of test cases be selected? What metrics can be used to select these tests?

The questions presented are closely linked to the fields of test case selection, prioritization, and test suite minimization. Ideas used in these areas are suspected to provide starting points for this work. Therefore, literature research into these areas is conducted.

1.2 Motivation

The main motivation for this work is to address the three problems mentioned. Addressing these problems could lead to the following benefits:

- Faster Time to Feedback: The goal is that a shorter response allows developers to find bugs in their implementation faster and increase their efficiency. The goal is that this quicker feedback can be used in two ways: The short run can be used to do a prerun, which ensures that the code has a high likelihood of being bug-free. For this scenario, the full pipeline runs if the prerun with the time constraint approach is successful.
- Reduce Computational Demands: The second way that the approach can be utilized is to employ time-budgeted runs during the day and schedule full pipeline during the night or when the demand for the CI system is low. The goal is to save computational resources.
- A Step Towards Green IT: The reduction of computation may lead to a reduced need for runners and less utilization of the runners installed. The consequence:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

fewer must be bought, and the installed runners require less power to execute computation. This leads to fewer resources needed to produce runners and a lower demand for power, which, if not produced with sustainable energy, reduces greenhouse gas emissions.

1.3 Expected Results and Research Questions

It is expected that the method will be able to reduce the time as the research that aimed to achieve something similar did succeed. However, the few papers published on this topic often come with limitations, like only applying to regression test [67] or not to new features [26].

This work will utilize three different methods where the goal is to figure out different insights into the research area:

1. Literature Research: To find possible implementation approaches, it is required to investigate approaches to reduce test suites already published. The main focus lies on Test Case Selection, Prioritization, and Test Suite Minimization Methods. Ideas from those fields are used for ideas to design the time-constraint method developed for this thesis. Relevant approaches include reducing the number of similar test cases [64] and the utilization of search strategies [2], [3], [67]

2. Design Cycle: The design cycle method [86] is used to develop the approach in iterations. The goal is to react to changes quickly to develop an approach that yields results comparable to running a full test suite in a reduced time budget.

3. Case Study: To test the approach's feasibility, a case study is conducted. An industrial project was selected because of the size of the codebase and the offer by the company to provide access to their CI resources. This is the reason the approach has not been tested with an open-source project. Although the project's codebase is used in the implementation phase to test the approach, the developed method is not project specific and should apply to any other codebase. (Even though this cannot be tested due to resource constriants.) Furthermore, it is expected that the large codebase, as well as the provided infrastructure, provide the opportunity to conduct a thorough investigation of the method's feasibility.

From a scientific perspective, the following research questions are addressed:

1. What characteristics and criteria should be considered in developing a test case selection algorithm that effectively determines the tests within a suite that are susceptible to the impact of pushed commit(s)?

2. Which of the determined tests are feasible and essential to execute within a given time constraint in order to maximize bug detection?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3. How many bugs in the code does such an approach find compared to running the whole test suite?

4. What is the relationship between the duration of the time budget and the effectiveness of a selected test set, and how can the optimal time budget be determined for achieving the best results?

It is expected that the following results will be achieved corresponding to the research questions:

1. To find a suitable approach, it is expected that literature research will present different approaches that rely on different metrics. It is suspected that some experimentation is required to satisfy the requirements.

2. To have the data at hand when deciding on which tests to fit into the time budget, it is suspected that the metrics need to be collected in a nightly pipeline to have them available at all times without computation effort.

3. To evaluate the effectiveness, the developed method will have to be executed with different commit sizes and multiple changes. The effectiveness is suspected to depend on the number of changes made during the commit. For this analysis, Random Testing (RT) will be utilized.

4. A universal recommendation for a time budget will most likely be impossible because every project is different. However, it could be that the 80/20 rule [61], also called the Pareto principle [80]. Other options are for the effectiveness to reach a plateau, decline after a fixed peak, or have no discernable correlation.

1.4 Structure

The complete structure of this thesis can be seen in the table of contents. The thesis started with the introduction (chapter 1). The introduction starts with the problem statement, presenting problems caused by long Continuous Integration pipelines. These issues serve as the foundation for the motivation part of the thesis. The third subsection explains what this thesis hopes to achieve. The current section provides an overview of the thesis's structure.

Chapter 2 supplies an overview of the fundamental building blocks required as the base for this work. It starts with the basics of testing. The first section is on the goals and economics of testing and how code coverage can be utilized. This is followed by an introduction to black-box and white-box testing, explaining the differences between these test types. A section about optimization problems follows, where the knapsack problem is introduced due to its importance for this work. The fundamentals chapter finishes with evolutionary algorithms and their core principles.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Related work (chapter 3) is focused on current research in the areas of test case selection and prioritization, as well as associating subjects. The chapter starts with information on coverage and which goals should be set when utilizing the metric. Then, current research on test case selection and test case prioritization is presented, followed by an overview of work that implements a time budget, as it is done with this work.

This is followed by the chapter discussing the methodology (chapter 4) where the scientific approach of this work is discussed.

This work's main focus is explained in the method design (chapter 5), which follows the design cycle methodology [86] and consists of 9 iterations plus an iteration 0 where the requirements are defined which specify the scope of the work. Each iteration consists of four different stages, though sometimes stages are combined if the description for a stage is short due to reduced complexity.

The implementation is followed by a chapter called evaluation (chapter 6), where the developed method is tested and evaluated. Afterward, the results are discussed (chapter 7) before this thesis concludes (chapter 8) and suggests possible shortcomings and ways to improve the approach.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Fundamentals

This chapter explains the fundamental terms and techniques essential for the "related work", "requirement analysis", "method design", and "evaluation" chapter. The first section is focused on the different kinds of software testing. The second section gives an overview of Continuous Integration/Continuous Delivery, then optimization problems and evolutionary algorithms are explained.

2.1 Testing

Software testing is crucial to software engineering and the broader software development landscape. According to G. O'Regan [63], software testing plays an integral part in demonstrating that a piece of code is working for the task it was designed to fulfill. Furthermore it shows that the functionality of the product under development is as desired. In the third edition of one of the classical books in this field, "The Art of Software Testing" [56], first published in 1979, stated that in 1979, software engineers knew the importance of software testing. According to the book, the rule of thumb was that about 50 percent or more of the total development cost was invested in testing. Singh [77] goes even further. He states that testing failures could threaten our civilization, which is highly dependent on the automated processes that software enables.

Therefore it can be argued that software testing is one of the most integral parts of the software engineering domain. This section recaps the fundamentals of this topic and serves as the foundation of this work. It starts with the goals and economics of testing. Then, it explains the difference between white-box and black-box-testing.

2.1.1 The Goals and Economics of Testing

The ultimate goal of software testing is: to find bugs in a software project and fix them [56]. Ultimately, however, it is impossible to find all bugs in software through testing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Not even trivial ones and especially not big software projects. They are never entirely bug-free [56].

With this in mind, the questions of how much to test and when to stop carry a lot of importance. These can either be approached from an economical viewpoint [81] or from a software engineering one. The question of where to set the goal is none that can be easily answered [54]. One metric often used, that is important for this thesis, is code coverage.

Code Coverage

Code coverage is an instrument, a metric, that software engineers employ to find untested code sections. It can be used automatically and is utilized by big companies like Google [38]. Meyers [56] distinguishes between five different kinds of coverage:

1. Statement Coverage: As a general rule, a statement is one instruction that a program instructs a machine to execute [92] This type of coverage is met when each statement of a program is executed exactly once. What is important to consider here is that the likelihood of triggering an error with this type of coverage is rather low [47].

2. Decision Coverage: A stronger criterion than statement coverage. Each decision must be evaluated as true or false at least once. Furthermore, this ensures that if there is an if-else in a program, the if and the else branches have to be executed at least once [56].

3. Condition Coverage: Condition Coverage is similar to Decision Coverage but different in that each condition part of a decision is met at least once [56]. Or with an example "IF A OR B", A and B should evaluate to true and false once [47]

4. Decision/Condition Coverage: Requires the combination of Decision and Condition coverage [56].

5. Multiple-condition Coverage: This type of coverage is defined as follows: "This criterion requires that you write sufficient test cases such that all possible combinations of condition outcomes in each decision, and all points of entry, are invoked at least once [56]."

2.1.2 Black-box vs White-box Testing

While rarely the term gray-box testing [20] is used in the context of software testing, usually literature [56], [63], [84] separates software testing into two sections: black and white-box testing.

Black-box testing, data-driven testing, and sometimes input/output-driven testing all regard the same thing. The idea is that the tester or test program has either no knowledge

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

or completely disregards his knowledge of the inner workings of the System Under Test (SUT) [56]. Moreover, as the name suggests, these tests are purely input and output-driven, meaning that some input is fed into the program, and the test verifies that the program gives some output. Therefore, the input for this testing is derived from the software specification [56]. The goal of black-box testing is to show that the functionality of the SUT, which could be a module, a feature, or a whole system, is as intended [63].

As the name suggests, white-box testing, sometimes also called logic-driven, follows the opposite idea to black-box testing. Where with black-box testing, there is no consideration of the internal structure of a program when writing a test, with white-box testing, the internal logic of the program is known, and tests are written with the program control flow in mind [63]. The goal is often to do so-called "exhaustive path testing"; this is where each path/branch of the program is executed. This should lead to full coverage of the program. Realistically, this can rarely be done since the number of paths in a program does increase exponentially the bigger the program becomes [56].

Kinds of Tests

After distinguishing between black-box and white-box tests, what is the methodology of testing? Which kinds of tests are there? Generally, tests are split into different categories, which include, but are not limited to:

1. Unit Tests: Unit or module tests focus on a small part of code. Larger programs require a more sophisticated testing strategy because they are too big to be tested as a single unit. Therefore, testing is split to ensure that a big program's smaller building blocks are tested thoroughly. These smaller blocks can be sub-programs, classes, (sub)procedures, or subroutines. This approach brings multiple benefits. It allows better focus, easier debugging, and parallelized testing when splitting the tests into modules [56].

2. Integration Test: This kind of test aims to verify that all modules with their corresponding interfaces work in tandem and aims to find problems in comparability. This is done after it is verified that each component works individually [63].

3. System Tests: As the name suggests, system tests test the system, specifically regarding the defined requirements. As this implies, however, this can mean a wide range of different specifications. A separate group typically does these tests and can, apart from the typical functional constraints, include various aspects such as security, software performance, and usability constraints [63].

4. UI Tests: Graphical User Interface tests test the Graphical User Interface (GUI) of a program. While these can be automated, in practice, they often are not, or not fully as there are still challenges with fully automating them, like identifying elements of the GUI and synchronization [59]. Regardless, there are techniques

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

to automate this kind of test like capture and replay [1], different model-based techniques [45], or artificial intelligence-based [44] ones.

2.2 Continuous Integration/ Continuous Delivery

Continuous Integration/Continuous Delivery (CI/CD) combined with a Source Code Management (SCM) system is fundamental to deploying the approach developed for this work. The SCM is needed to keep track of changes; therefore, it is the key to finding tests linked to changes. Moreover, CI/CD is mandatory to automatically execute shortened runs and especially to keep the database used to store coverage information on tests up to date.

How important the concepts of CI/CD are in software engineering cannot be overstated. For example, Shahin, Ali Babar, and Zhu [75] open their article on the topic with the following statement:

> Continuous practices, i.e., continuous integration, delivery, and deployment, are the software development industry practices that enable organizations to frequently and reliably release new features and products [75].

The topic of CI/CD is a highly researched area. Therefore, this section on the topic starts with a definition of the terms before introducing a general overview of how a pipeline can be used, and the implementation of the concepts will be described [75].

2.2.1 Definition

Generally, the term Continuous Integration/Continuous Delivery (CI/CD) can be split into the sub-parts of Continuous Integration as well as Continuous Delivery; however, according to Fitzgerald and Stol [29], there are other topics associated. This work will define CI/CD and associated topics like it is done by the two authors[29]:

- Continuous Integration: This aims to install an automated process within a project that repeats tasks such as compiling source code, verifying code quality, and adherence to standards. Of particular importance is the regularity with which these tasks are scheduled. Furthermore, a failure of the automated process should allow problems to be caught and resolved quickly [29], [46], [69], [74], [79].
- Continuous Delivery: As the name suggests, the counterpart to Continuous Integration with the goal to put the build software onto an environment. This can be already accessible by the end-user but is not mandatory [29], [60].
- Continuous Testing: Automated testing of software and its components. The goal is to detect errors introduced by a code change to fix them [8], [29].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Continuous Deployment: This means deploying software changes directly to customers automatically [19], [29], [66].

2.2.2 General Pipeline Overview

The overview of a pipeline used for Continuous Integration/Continuous Delivery varies depending on the concrete implementation and the steps. However, a generalized overview can be seen in Figure 2.1. This image was taken from Fitzgerald and Stol [29] (they quote two websites for the graph [1,2])

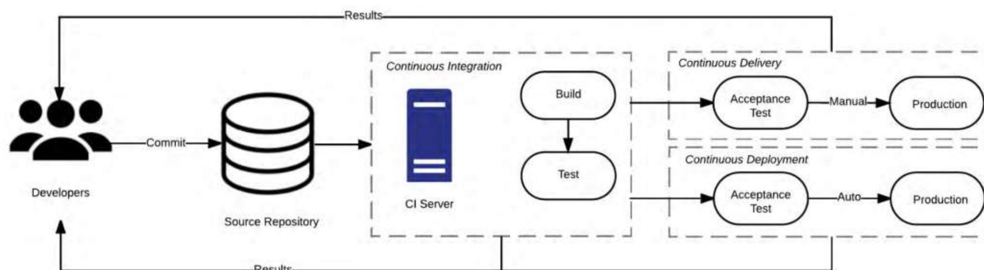


Figure 2.1: Overview of CI/CD pipeline taken from [29]

As presented in the image [29], the CI/CD pipeline is triggered through the developers by pushing a commit onto the Source Code Management (SCM) system. This starts the Continuous Integration section, illustrated in the image by first the build, where the software is compiled. The build code is then utilized to run tests automatically. As discussed in the testing sections, these can be of different types. Continuous Delivery and possibly Continuous Deployment will run Acceptance Tests before it can either manually or automatically be deployed to production.

2.3 Optimization Problems

Karsten Weicker [85] states in his book that optimization problems are unavoidable in almost all areas of society. They are present in various circumstances ranging from industrial applications to economic endeavors to scientific research questions. Examples of optimization problems include the question of "How can resources be utilized most effectively?" or one of the most famous being the traveling salesman problem. The traveling salesman problem represents cities and roads and the question in which order the salesman should visit each city so that the time he or she spends on the road is minimized.

[1]http://blog. crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuousdeployment, accessed on: 17.07.2024

[2]http://www. mindtheproduct.com/2016/02/what-the-hell-are-ci-cd-and-devops-acheatsheet-for-the-rest-of-us/, accessed on: 17.07.2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Generally Weicker [85] defines optimization problems as the search for a global optimum, which is described as follows:

$$X = x \in \Omega | x' \in \Omega : f(x) \geq f(x')$$

Where Ω is the search space, X is the global optimum part of Ω, and f is a Mapping from Omega to $\mathbb{R}$.

An optimization problem is generally a question where a solution with the biggest gain is searched. One important problem in this class is the knapsack problem.

2.3.1 Knapsack Problem

The knapsack problem is defined as follows [50]: Suppose there is a knapsack; this knapsack has a capacity, denoted with M. Furthermore, there is a range of objects. Each of these objects has a weight and a value. In total, there are n objects to choose from. The goal is to determine which objects shall be selected so that the value of the selected objects is maximized and the combined weight of the selected objects does not exceed the capacity M of the knapsack.

Generally, the knapsack problem is an NP-complete problem [42]. This indicates the difficulty in solving this problem for big instances.

2.4 Evolutionary Algorithms

Optimization problems can become quite hard in terms of complexity. Good examples of this kind of hard issue are different NP-complete problems, like the already mentioned knapsack problem. Generally, scientists/software engineers aim to find creative ways to deal with such problems if confronted. As stated by Weicker [85] one idea is to utilize the phenomenon of natural evolution. Natural evolution results from different factors like natural selection, mutation of genes, and recombination of genomes.

Generally, when solving an optimization problem, the ultimate goal is to reach the global optimum. The aim is to accomplish this faster than by comparing every permutation. Still, due to time constraints, finding the global optimum is often not feasible; here, evolutionary algorithms come in handy since their goal is usually to find a good solution in a fraction of the time needed to find the optimal solution.

There are a variety of different models that utilize the idea of evolution and are combined in the term evolutionary algorithms [78]. How an evolutionary algorithm can be implemented is shown in Figure 2.1.

2.4.1 Variation vs Selection vs Recombination

Generally, it can be said that two factors are influencing each other to achieve a successful implementation of an evolutionary algorithm. This is the interplay between variation, also known as mutation, and selection, which represents the idea of natural selection [85].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
procedure EA;
t = 0; /* Initial Generation */
initialize_population(t);
evaluate(t);
until (done) {
        t = t + 1; /* Next Generation */
        select_parents(t);
        recombine(t);
        mutate(t);
        evaluate(t);
        select_survivors(t);
}
```

Figure 2.2: General outline of evolutionary algorithm by Spears [78]

In Weickers book [85], variation is also called mutation. As the name suggests, this concept is derived from mutation genes. Here, the goal is to change small bits of the proposed solution to achieve a better result. In evolutionary algorithms, one usually talks about different generations where mutation is used from one generation to the next. In contrast to this stands selection. Selection is used to exclude results or different directions in mutation.

Compared to selection and variation, there is also recombination [78]. This is similar to mutation in that it is used in the "evolutionary" step, or more precisely, in the candidate creation process from one generation to the next. For example, if there is an encoding AABB with high fitness, the goal is to preserve the encoding part that yields the high fitness value and only mutate the rest. One strategy would be to split the part in the middle and then keep the part AA to create the next generation and only mutate the BB part. However, compared to mutation, where each part of the encoding/genome might be changed when recombining, keeping parts to preserve and increase fitness is desired.

2.4.2 No Free Lunch Theorem

The no-free lunch theorem describes the differences and improvements between general-purpose evolutionary algorithms and more optimized ones. Investigations into this field date some years back. For example, one framework for comparing such algorithms goes back to an article from Wolpert and Macready [88] published in 1997. In their article [88], they concluded with an important remark: "Roughly speaking, we show that for both static and time-dependent optimization problems, the average performance of any pair of algorithms across all possible problems is identical [88]."

This means that there is no evolutionary algorithm that should perform better in every problem instance. If an evolutionary algorithm is better optimized for one problem group, it will perform worse on another [78]. Furthermore, this implies that elements

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

like recombination can perform better in some situations, for example, if the number of sub-populations has a huge impact on the algorithm's performance but can, in others, hurt performance.

2.4.3 Genetic Algorithms

A genetic algorithm is a technique that can be applied to search and optimization problems. Like a normal evolutionary algorithm, the main goal is maximizing fitness, which correlates with minimizing loss [34]. Its main characteristic is its selection of parents for the next generation, which is based on probabilistic factors. Another common technique is recombination. With these two aspects, the role of mutation is often lower compared to classical evolutionary algorithms [85].

Generally, to implement a genetic algorithm, the simplest method is to encode the population which is the search space of possible solutions as a binary string [85]. If we go back to the knapsack problem discussed with the basics of optimization problems, it would be possible to encode selected objects as a 1 and objects that shall be left out of the final solution by 0. As Weicker [85] states, binary encoding is rarely sophisticated enough to encode problems since most instances do not come down to a simple yes or no question. For this work, however, this encoding is sufficient due to the properties of the knapsack problem and that said encoding can be translated to a knapsack instance.

Two different fundamental techniques function as a starting point for genetic algorithms. These are:

1. Standard Genetic Algorithm: A standard genetic algorithm is classified by one main attribute: the complete regeneration of the population after each generation. This means that after each generation, the parents are not used anymore, and the complete generation is new [85].

2. Steady-State Genetic Algorithm: Compared to a standard algorithm, the steady-state algorithm's implementation does not throw the whole population away after each iteration, but the generations' populations overlap. After each generation, only one new element is added to the population, and only its parent gets replaced [85].

Figure 2.2 was taken from Weicker's book [85] and illustrated the different schemes of both approaches. Figures 2.3 and 2.4 show concrete implementations of a genetic and a steady-state genetic algorithm.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

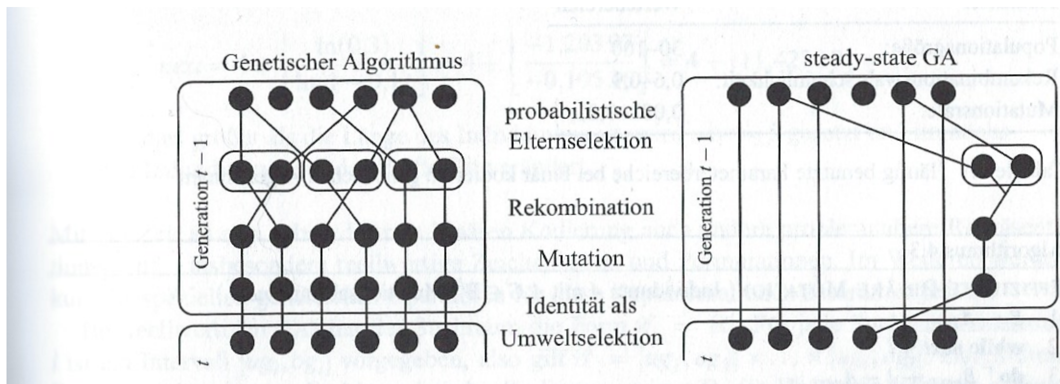


Figure 2.3: GA vs steady state GA [85]

GENETISCHER-ALGORITHMUS(Zielfunktion F)

1 $t \leftarrow 0$
2 $P(t) \leftarrow$ erzeuge Population mit μ (gerade Populationsgröße) Individuen
3 bewerte $P(t)$ durch F
4 **while** Terminierungsbedingung nicht erfüllt
5 **do** ⌜ $P' \leftarrow$ Selektion aus $P(t)$ mittels SELEKTION-FITNESSPROPORTIONAL
6 (Es sei: $P' = \langle A^{(1)}, \ldots, A^{(\mu)} \rangle$)
7 $P'' \leftarrow \langle \rangle$
8 **for** $i \leftarrow 1, \ldots, \frac{\mu}{2}$
9 **do** ⌜ $u \leftarrow$ wähle Zufallszahl gemäß $U([0, 1))$
10 **if** $u \leq p_x$ (Rekombinationswahrscheinlichkeit)
11 **then** [$B, C \leftarrow$ EIN-PUNKT-CROSSOVER$(A^{(2i-1)}, A^{(2i)})$
12 **else** ⌜ $B \leftarrow A^{(2i-1)}$
13 ⌞ $C \leftarrow A^{(2i)}$
14 $B \leftarrow$ BINÄRE-MUTATION(B)
15 $C \leftarrow$ BINÄRE-MUTATION(C)
16 ⌞ $P'' \leftarrow P'' \circ \langle B, C \rangle$
17 bewerte P'' durch F
18 $t \leftarrow t + 1$
19 ⌞ $P(t) \leftarrow P''$
20 **return** bestes Individuum aus $P(t)$

Figure 2.4: Genetic algorithm pseudo-code [85]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
STEADY-STATE-GA( Zielfunktion F )
 1  t ← 0
 2  P(t) ← erzeuge Population mit μ (Populationsgröße) Individuen
 3  bewerte P(t) durch F
 4  while Terminierungsbedingung nicht erfüllt
 5  do ⌈ ⟨A,B⟩ ← Selektion aus P(t) mittels FITNESSPROPORTIONALE-SELEKTION
 6        u ← wähle Zufallszahl gemäß U([0, 1))
 7        if u ≤ p_x (Rekombinationswahrscheinlichkeit)
 8        then ⌊ C ← EIN-PUNKT-CROSSOVER(A,B)
 9        else ⌊ C ← B
10        D ← BINÄRE-MUTATION(C)
11        bewerte D durch F
12        P' ← entferne das schlechteste Individuum aus P(t)
13        t ← t+1
14     ⌊ P(t) ← P' ∘ ⟨D⟩
15  return bestes Individuum aus P(t)
```

Figure 2.5: Steady state genetic algorithm pseudo-code [85]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 3

Related Work

After the fundamentals, which are the essential building blocks of this work, this chapter will focus on current research in test case selection and prioritization as well as areas that influence that field or are of interest to this work.

3.1 Coverage Goals

As this work is about determining which test cases are important for the changed code and finding good coverage for the rest of the code base, an important question is: Which coverage goals should be aimed for? Another question is closely linked: Which coverage data should be used to maximize efficiency and reach the objective?

As described in chapter 2, there are different kinds of code coverage. This raises the question of which type of coverage will produce a satisfactory indication of effectiveness for testing. Some insights into this issue was presented by Wei, Meyer, and Oriol in their paper from 2012 [54]. They asked, "Is Branch Coverage a Good Measure of Testing Effectiveness?" To answer this question, they researched how branch coverage evolves when employing random testing to 14 classes for 2520 hours while tracking how branch coverage changed and which faults were not found.
According to the paper, the results showed that, at first, there was a strong correlation between the branch coverage and the number of faults found. However, this was only within the first 10 minutes of testing, which is a small portion of the time compared to the amount they spent on randomized testing. Furthermore, about 50 percent of the failures were uncovered at a point where branch coverage barely changed. As the authors concluded, this implies no strong correlation between branch coverage and the failure detection rate. This demonstrates that branch coverage is not the most effective measure for coverage.

	Chart	Closure	Math	Time	Lang	**Total**
Total # of faults	26	133	106	27	65	357
# of CodeCover/DUAF problematic cases	0/1	0/23	0/45	0/0	14/0	14/69
Total # of studied faults	**25**	**110**	**61**	**27**	**51**	**274**
	% of detected faults					
Statement	32%	5%	10%	0%	14%	10%
Branch	32%	18%	18%	11%	18%	19%
MC/DC	24%	18%	18%	11%	25%	19%
Loop	12%	5%	18%	0%	8%	8%
All Control-flow	**44%**	**24%**	**33%**	**15%**	**29%**	**28%**
# of undetected faults by control flow criteria	14	84	41	23	36	198
	% of detected faults					
def-use (DUA)	86%	87%	80%	91%	50%	79%
Data & control-flow	**92%**	**90%**	**87%**	**92%**	**65%**	**85%**

Figure 3.1: Summary of the coverage criteria effectiveness in five subjects of study from [43]

This raises the question of what an effective coverage metric is. One paper from 2018 [43] evaluated five different coverage metrics and investigated their ability to help the authors identify faults in open-source projects. The authors compared a mixture of control flow and data flow coverage. Control flow coverage allows the identification of uncovered parts in the execution flow of a program, while data flow coverage allows the examination of which values are assigned to variables and how they affect the program. More precisely, they investigated the effectiveness of Statement Coverage, Branch Coverage, Modified Condition-Decision Coverage (MC/DC), Loop coverage, and def-use pair coverage.
They observed that no single metric could identify 100 percent of all faults. Generally, the control flow coverage performed worse than the data flow coverage. No single metric performed best on its own; however, the best performance resulted from combining these different metrics. Then, on average, 85 percent of faults were found. To find the remaining 15 percent of faults, the authors suggest that specification-based testing could help increase that number. Figure 3.1 was taken from the article [43] and shows the results the authors found. The two plots were generated to illustrate the averages of the different methods and the individual results on the different projects.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

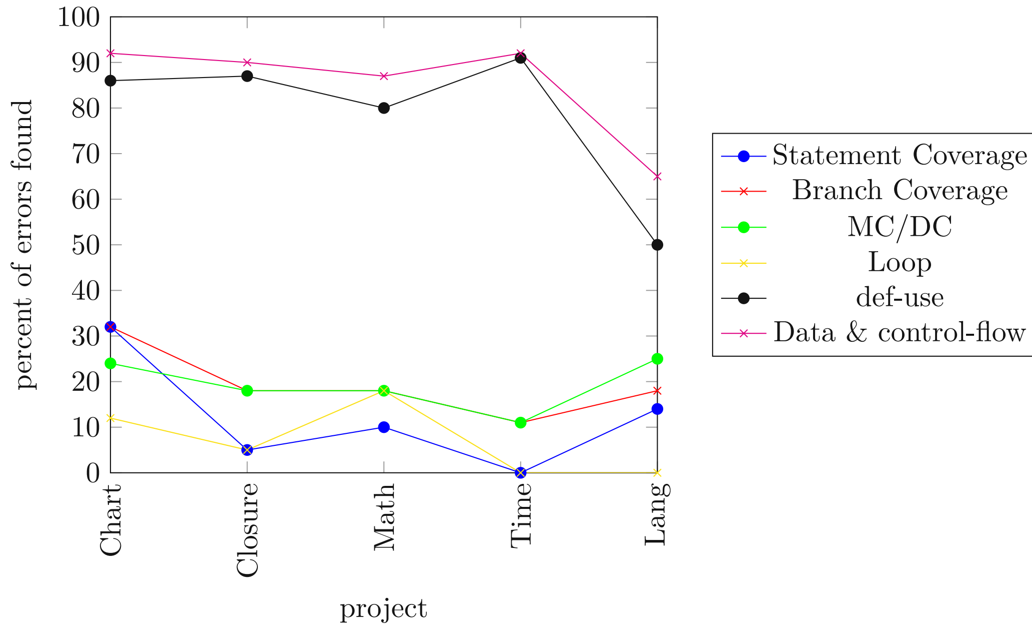


Figure 3.2: Percent of errors found per project [43]

3.2 Test Case Selection and Prioritization

Two research areas that overlap with this work are test case selection and test case prioritization. Therefore, the research shown in this section concentrates on selecting test cases as well as test case prioritization techniques. Test case prioritization aims to get test suites to fail fast or, phrased differently, get tests to fail early. Regardless of the technique employed, at least one criterion is used to order the test cases. However, different metrics can be used, like coverage, the history of the code base, or requirements [89].

In comparison to test case prioritization, in test case selection, a new system version is compared to an older version. The goal is to select the tests related to the new version or ones that might be impacted by the changes [70]. Test case selection techniques can be one of two things: safe or unsafe. Safe test case selection techniques select everything that has the potential to reveal at least one fault. Unsafe techniques, however, do not guarantee this. They select all test cases that are modification-traversing. This can lead to similar fault detection rates but usually at a lower cost [32]. As mentioned by Di Nardo, Alshahwan, Briand, and Labiche [58], generally, with test case selection huge gains in time can be made, some papers show a reduction up to 98 percent [30], [33], [71], [73]. However, the gains can be as low as zero percent [33].

The general goal of these areas is to reduce the time needed to execute the test suite. Usually, there is no time box, which is the main distinction between their approach and this work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

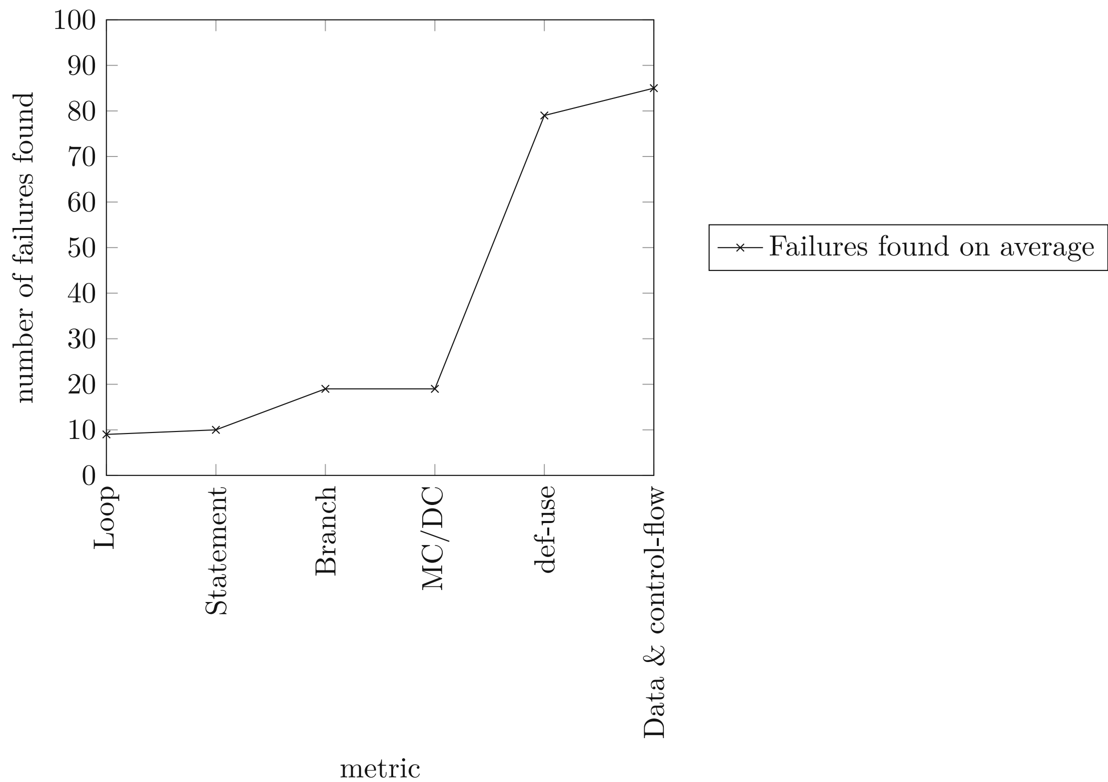


Figure 3.3: Average percent of errors found per metric [43]

Different techniques can be utilized to implement test case selection and prioritization. A wide number of articles focus on this topic, of which some compare different techniques like Engstrom, Runeson, and Skoglund [24], as well as Yoo and Harmon [89] did for regression-testing, or test suite selection [25]. Henard et al. [37] compare test case prioritization techniques, splitting the techniques into two groups: white-box and black-box prioritization. To provide a better overview, this work will also explain some techniques that can be utilized.

3.2.1 Coverage-based Techniques

For white-box prioritization techniques, the first and according to Hennard et al. [37], the most common set of techniques are the ones that employ dynamic coverage information. So, this part inspects different approaches where coverage information is the main metric used for selection or prioritization.

The first article found when conducting literature research for this work was from Sinaga [76], where he focuses on comparing four different test case prioritization techniques, which he compares against random testing. Firstly, Random Testing (RT), then ART (Adaptive Random Testing) utilizing the Manhattan distance, then addition plus ART

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

and ART plus additional. The author [76] describes the methods as followed:

- Random Testing (RT): Used as a benchmark, the test cases to be executed are randomly selected.
- Additional: This technique, as suggested by Rothermel et al. [72], selects test cases according to which test cases add the most additional branch coverage.
- Adaptive Random Testing (ART): The ART test case prioritization technique Sinaga [76] investigated as it was proposed by Zhou [91]. It improves upon Random Testing by adding fault detection, which utilizes information about the location of test cases already executed. In the paper by Zhou [91] it is mentioned that ART had quite a large number of publications look into the method [12], [13], [14], [15], [16], [18] with one even calling it "one of the most effective approaches in the automatic test generation area" [40]. The specific implementation used [76] adapts a Fixed-Size Candidate Set (FSCS), which consists of ten candidates. The difference between the candidates is calculated with the Manhattan distance of the different test cases' branch coverage.
- Additional with ART: As the name implies, this method combines Additional with ART. This is done by utilizing additional until the test set covers all branches, and then ART takes over to order the remaining test cases.
- ART with Additional: Just like the previous method, this one also combines Additional with ART, but the difference here is ART is employed in every step of the selection process; however, the Manhattan distance is not used; instead, the distance is calculated using calculations employed in the additional method.

According to the results of Sinaga [76], combining Additional and ART yields better results than Random Testing or both methods separately.

The next paper that looked at coverage-based techniques was written by Di Nardo, Alshahwan, Briand, and Labiche [58]. It was designed as a case study comparing different test-case selection techniques. They tested four different test case prioritization techniques, which they deployed to four different projects. The authors compared the methods to Random Testing to get a reference. Furthermore, they investigated a test case minimization technique and a hybrid form of test case prioritization and minimization. The different techniques are described [58] as follows:

- Total Coverage: For this technique, the only criteria considered is how much of the total coverage the specific test at hand makes out to be. If it is imaged as a percentage of the total, the test with the higher percentage is selected first. If two tests cover the same code, they will be selected randomly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Additional coverage: The additional coverage is the same as was described before in the article of Sinaga [76]. Here, the test that adds the highest coverage is selected.
- Total coverage of modified code: As the name suggests, this technique works almost like the total coverage technique, but in comparison to just looking at all code that changed, it only considers code that has been modified.
- Additional coverage of modified code: Just like the technique before was to total coverage, this one is to additional code. It considers which test adds the most coverage, but instead of considering the coverage for the whole code base, it just considers coverage of modified code

The authors [58] found out that coverage criteria that rely on a finer granularity achieve better results than the ones that are more coarse. This was especially true for additional coverage, which in testing beat out total coverage on a confidence level of 95%. Furthermore, it showed that techniques based on modified code did not perform better than their regular counterparts. For test minimization techniques, coverage techniques on the coarser side had a better minimization potential than those relying on finer-grained methods. The authors discovered that hybrid methods exceeded the results of traditional approaches, particularly in reducing the test suite size. They found that the hybrid approach did not improve results enough to justify itself.

3.2.2 Similarity-Based Techniques

Another technique, regularly used for optimizing test suites is based on similarity or distance functions [36], [48], [62], [65]. One approach presented by o G. de Oliveira Neto, Ahmad, Leifler, Sandahl, and Enoiu [64]. They conducted a case study where, for two test cases, they determined the similarity between all tests, resulting in a similarity matrix. One value on this matrix is calculated as the similarity between tests a and b. They described the similarity value as 1 if the tests are 100 percent similar or 0 if they have nothing in common. Otherwise, it is in between. The authors [64] state that some studies [27], [28] instead use the term distance, which is just calculated as $distance = 1 - similarity$, this means that 0 means the tests have no distance and 1 means the tests are 100 percent distinct.

Generally, for similarity-based test case selection, one of the most crucial decisions when implementing the test case selection process is which properties to compare. For this modification-information [65] as well as information about failure history [62] and textual steps [10], [11] can be used. It makes a difference with which function the similarity is calculated. Different methods exist, which researchers tested in different studies [21], [35]. For their case study, the authors [64] compared three different functions for a case study to use similarity matrices for test case selection. The three different methods were:

1. Jaccard Index: The Jaccard Index was invented by Jacob Jaccard and his work "The Distribution of the Flora in the Alpine Zone" [39] published in 1912. Originally,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

it was described as

$$\frac{\text{Number of species common to two districts}}{\text{Total number of species in the two districts}} * 100$$

by Jaccard [39].

2. Normalised Levenshtein: The concrete implementation of this function was not stated in the article [64]. However, the IEEE paper "A Normalized Levenshtein Distance Metric" [90] by Yujian and Bo from 2007 described how it functions. The objective is to calculate the distance between two strings, X and Y. This is done by "a simple function of their lengths (|X| and |Y|) and the Generalized Levenshtein Distance (GLD)" [90]. The paper describes two versions of the Normalized Levenshtein Function. These are denoted as NED_1 and NED_2 and are defined as follows:

$$NED_1(X,Y) = min\left\{\frac{W(P_{X,Y})}{L(P_{X,Y})}\right\}$$

$$NED_2(X,Y) = min\left\{\frac{W(P_{X,Y})}{|X| + |Y|}\right\}$$

X and Y are, in this case, strings over an alphabet, and the function of $L(P_{X,Y})$ was quoted by the authors [90] to be from other works [4], [5], [52], [83] and is the "number of elementary edit operations described by $P_{X,Y}$" [90]. The generalized Levenshtein distance is defined as follows:

$$GLD(X,Y) = min\{W(P_{X,Y})\} = min\{\gamma(T_{X,Y})\}$$

With $P_{X,Y}$ as a path from X to Y containing the edits needed from X to Y (editing path). The last formula defines the edit transformations weight γ of $P_{X,Y}$ which is stipulated as followed:

$$\sum_{k=1}^{L(P_{X,Y})} \gamma(T_i)$$

Furthermore, $T_{X,Y}$ is exactly this weight.

3. Normalized Compression Distance: This technique was taken from one paper [28] which conducted an analysis with it on the quantification for test sets. Normalized Compression Distance (NCD) uses concepts from Information Theory like information distance, which the authors [28] calculated with the following formula:

$$ID(x,y) = mac\{K(x|y), K(y|x)\}$$

Here, x and y denote two strings. $K(x)$, which can be seen in the formula, is the Kolmogorov complexity [49]. It can be represented in bits and denotes how long a program has to be which outputs a given input, which in the example is x. Originally, it was proposed by Bennet et al. [7] to utilize Kolmogorov complexity

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

to calculate the information distance. There are some issues with the Kolmogorov complexity, especially because longer strings usually have a bigger information distance than shorter ones [28]. To address this, the Normalized Information Distance (NID) idea proposed by Li et al. [55] was used and looks as follows:

$$NID(x,y) = \frac{max\{K(x|y), K(y|x)\}}{max\{K(x), K(y)\}}$$

However, due to practicability reasons, two authors [17] proposed an approximation, the Normalized Compression Distance, which was also the one used in the analysis of this method [28] and in the comparison of the three functions [64]. The approximation [17] looks as follows:

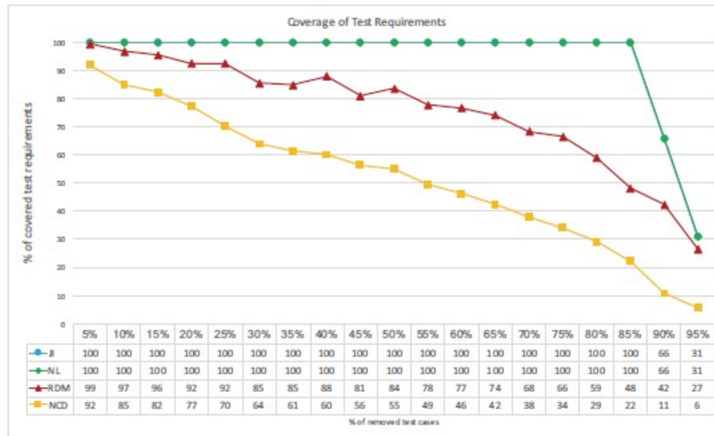
$$NCD(x,y) = \frac{C(xy) = min\{C(x), C(y)\}}{max\{(C(x), C(y)\}}$$

Here, x and y are two strings, and C(x) represents the length of the compressed version of x.

Generally, the test suite was downsized in increments of 5 percent, using different methods and random test case selection to verify the results. The authors conducted experiments on two different companies. For the first company, they presented the "coverage of test requirements" as "coverage of test dependencies" and as the "time reduction". For the second company they showed "coverage of test steps". The resulting graphs from the paper can be seen in Figures 3.2 and 3.3 and were directly taken from the article [64].

As can be seen in Figure 3.2, when "coverage of test requirements (a)" and "coverage of test dependencies (b)" are compared, Normalised Levenshtein (NL) as well as Jaccard Index (in graphs as JI) outperform the Random Selection (RDM) and Normalized Compression Distance (NCD). In terms of time, this factor decreases quasi linearly for all methods. For the second company, the results were quite similar. Generally, it can be seen that NCD performed quite poorly, in some cases worse than the random control.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
JI	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	66	31
NL	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	66	31
RDM	99	97	96	92	92	85	85	88	81	84	78	77	74	68	66	59	48	42	27
NCD	92	85	82	77	70	64	61	60	56	55	49	46	42	38	34	29	22	11	6

(a)

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
JI	100	100	100	100	100	100	100	99	99	99	99	99	99	85	71	57	43	28	14
NL	100	100	100	100	100	100	100	100	100	100	100	100	100	85	71	57	43	28	14
RDM	97	96	92	92	88	79	78	77	71	68	63	61	59	52	44	36	30	23	13
NCD	90	81	73	65	59	54	51	47	42	39	33	27	24	20	15	11	9	8	4

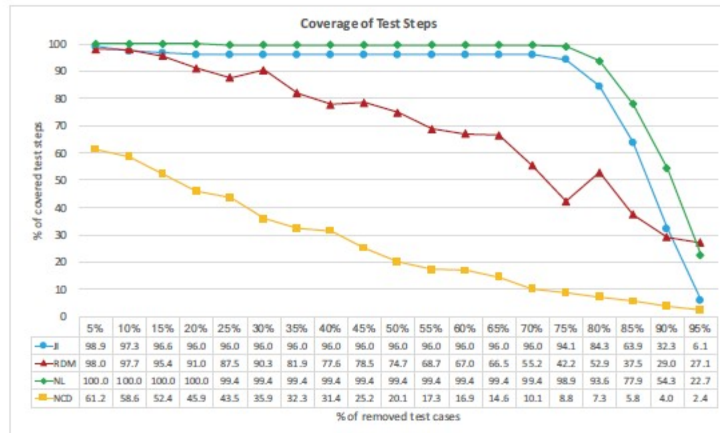
(b)

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
JI	208	205	196	185	179	173	168	164	134	118	108	105	87	84	59	36	19	14	6
NL	189	181	158	142	140	138	131	125	106	86	60	54	44	34	28	24	17	13	7
RDM	198	195	185	161	150	147	131	130	116	102	101	70	75	70	52	40	24	18	11
NCD	202	197	196	182	174	165	158	140	125	120	93	87	80	75	60	55	51	33	24

(c)

Figure 3.4: Evaluation of first company [64]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
JI	98.9	97.3	96.6	96.0	96.0	96.0	96.0	96.0	96.0	96.0	96.0	96.0	96.0	96.0	94.1	84.3	63.9	32.3	6.1
RDM	98.0	97.7	95.4	91.0	87.5	90.3	81.9	77.6	78.5	74.7	68.7	67.0	66.5	55.2	42.2	52.9	37.5	29.0	27.1
NL	100.0	100.0	100.0	100.0	99.4	99.4	99.4	99.4	99.4	99.4	99.4	99.4	99.4	99.4	98.9	93.6	77.9	54.3	22.7
NCD	61.2	58.6	52.4	45.9	43.5	35.9	32.3	31.4	25.2	20.1	17.3	16.9	14.6	10.1	8.8	7.3	5.8	4.0	2.4

Figure 3.5: Evaluation of second company [64]

3.2.3 Search-Based Techniques

Another set of techniques used for test case prioritization and selection are search-based techniques. These are used by different authors for different purposes [2], [3], [67]. One paper to present in this regard is by Arrieta, Wang, Sagardui, and Etxeberria [2] and utilized search-based strategies for test case prioritization in the context of Cyber-Physical Systems (CPS).

One concept is that of feature models. Feature models are tree-like structures that allow to break down product lines and visualize similarities [6]. In Figures 3.4 and 3.5 an example from Batory [6] illustrates how such a feature model can look. These feature models are crucial for the work of Arrieta, Wang, Sagardui, and Etxeberria [2]. Generally, to represent the CPS product line, a feature model and a configuration file are provided to a requirements parser.
Then, this parser associates the input to the requirements of the product's features to be tested. Then, a search algorithm is utilized to prioritize tests associated with the requirements, thus reducing the needed time scope. The search algorithm includes the data from the parser, information from the test history, and information about user preferences. An overview of this process can be seen in Figure 3.6, which was taken from the article [2].

Within the search function itself, the authors [2] implemented four different search functions and compared them against random search as a benchmark. The results were that local search algorithms outperformed global search algorithms.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Figure 3.6: Feature diagram notations [6]

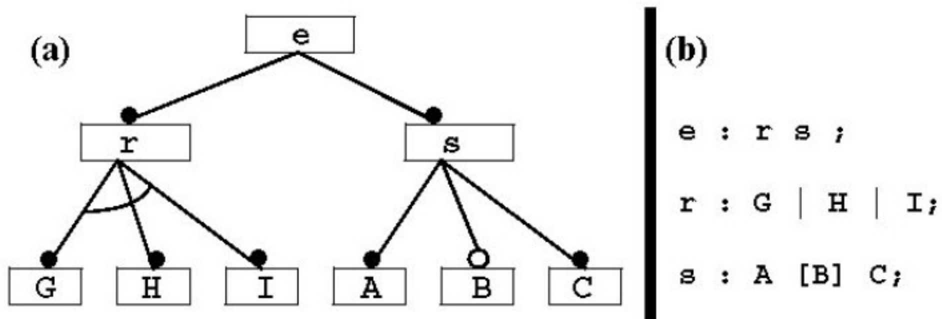


Figure 3.7: A feature diagram and its grammar [6]

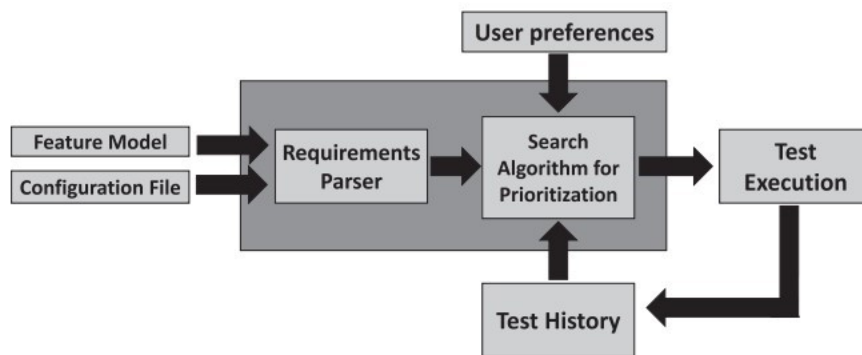


Figure 3.8: Overall overview of the proposed approach for test case prioritization. [2]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Another approach that utilizes local search for test case prioritization was presented by Lu et al. [51]. Like other work before it [9], they utilized so-called Ant Colony Optimization (ACO), which is based on an Ant Colony System (ACS). The idea of an Ant Colony System goes back to the last century when Dorigo and Gambardella [22] presented such a system as a possible solution for the Traveling Salesman Problem. Figure 3.7, taken from Dorigo and Gambardella [22], illustrates how they envisioned fundamentals of the ACS algorithm. The idea is that ants are simulated to select one route if they are at an intersection randomly. Each time they are on the way, the ant leaves pheromones behind. Because the lower route in the figure is shorter, it will be visited more frequently and, in the end, will have more pheromones attached.

Now that it is clear what ACS is, how does the approach by Lu et al. [51] function? The approach presented by the authors called Coverage-Based Ant Colony System (CB-ACS) can be seen in figure 3.9 which was taken from the article [51]. Generally, this CB-ACS aims to maximize statement coverage. According to the authors [51] it is possible to express this problem as an optimization problem with the following formula:

$$minimize \sum_{t=1}^{n} \Phi(t_i)$$

t_i denounces the test at position I from a test suite, and $\Phi(t_i)$ gives the value of the additional coverage when the test t_i is executed. First, this problem must be converted to a graph so that an ACS algorithm can solve it. The conversion to a graph (V, E), with V being the vertices and E the edges works as follows:

1. Each test of the test suite gets assigned to a vertex V_i. Furthermore, one vertex is added, with no tests assigned v_{-1}. This vertex is added as a starting point for the ants because the first test execution usually greatly influences the overall coverage. To avoid this and let the ants decide for themselves with which test case to start, the virtual vertex v_{-1}

2. There are two parts to the edge creation process. Firstly, each vertex gets connected to v_{-1} with an edge $e_i = (v_{-1}, v_i)$ Secondly, the original vertices v_i (each representing a test) all get connected to each other with an unidirectional edge.

3. Each vertex then holds the coverage information for its corresponding test.

The resulting graph can be seen in Figure 3.8, which was taken from the paper [51]. The remaining part is primarily the ACS algorithm. Some more improvements, like an Addition Coverage-Based Heuristic (ACB) and some other improvements, were also introduced. When comparing their approach to a benchmark their results seem promising and are out-competing with other state-of-the-art techniques, according to the authors [51]. Also, they state that experimental real-world results seem promising.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

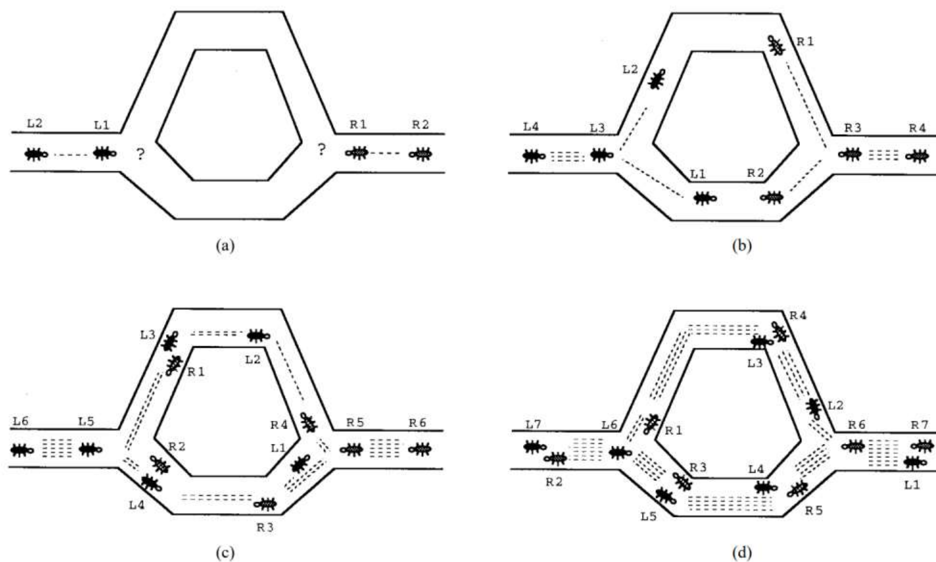


Figure 3.9: Example of an ant colony system for Traveling Salesman Problem [22]

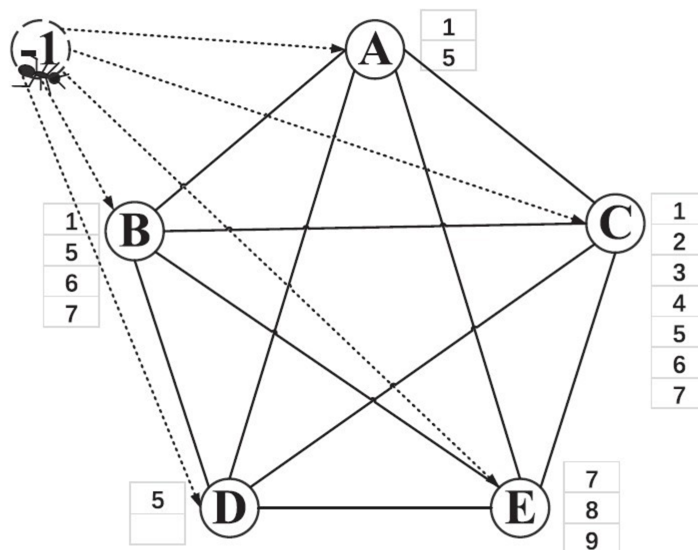


Figure 3.10: Graph for Coverage-Based Ant Colony System (CB-ACS) [51]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

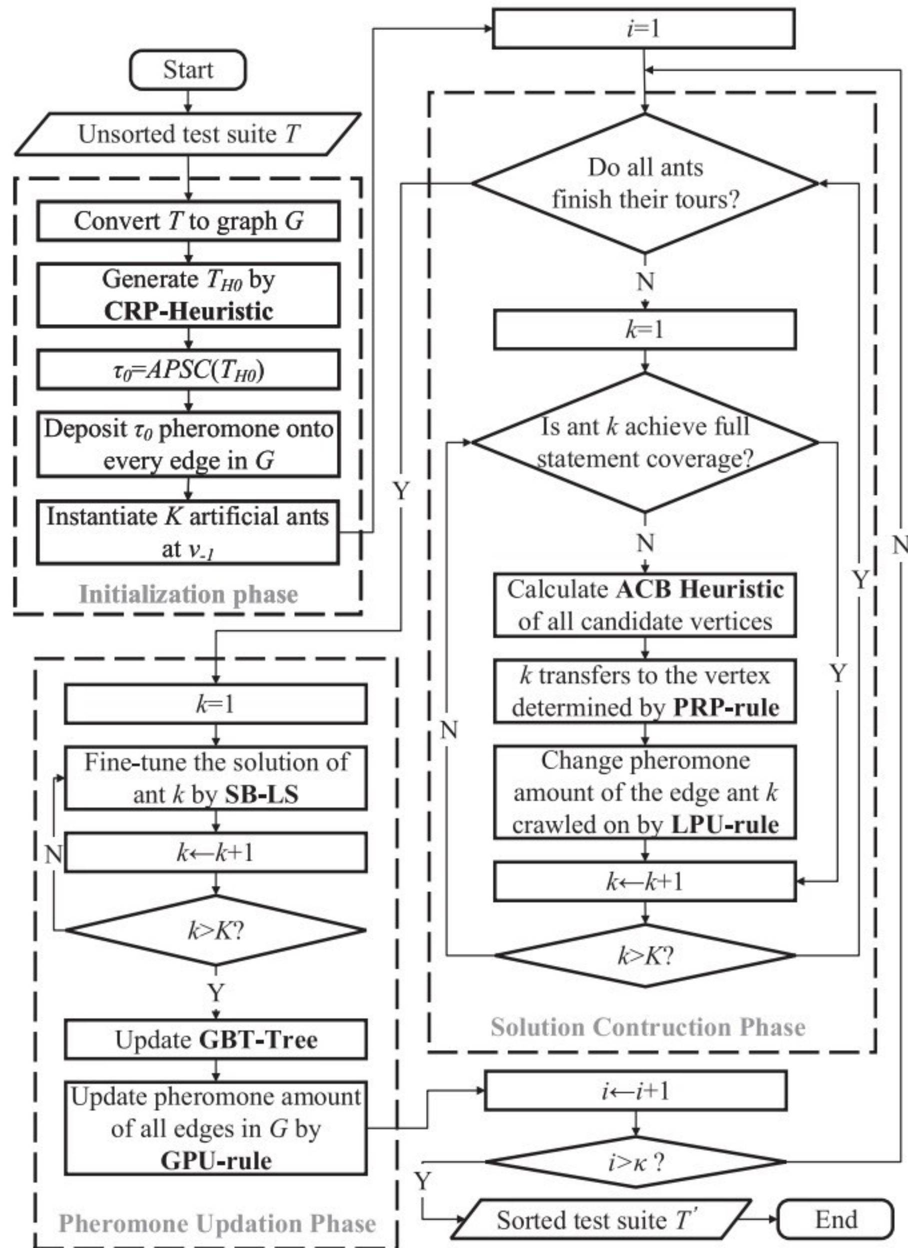


Figure 3.11: Overall overview of the proposed approach for test case prioritization [51]

3.3 Test Case Selection With Time Budget

Test case selection, which utilizes a time budget, is a rarely covered topic in research. Only one article was found during research that covered this topic when researching for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

this thesis. This could be because of the similarity to test case prioritization; however, it is distinct. The only paper to have covered that topic is from Farzat and de O. Barros [26] and focuses on the selection of unit tests.

The authors [26] reason to tackle this area is to reduce the time in the development process which is required for testing since this is well known to make up to 50 percent of the total time spent on a project [31].

They [26] implemented their approach with a heuristic and a local search approach to run tests within a time frame. Furthermore, they conducted a case study comparing the two approaches against random testing. For their testing, they restricted themselves to unit tests as defined by Myers [57] to verify software very fine-grained, allowing them to test specific program modules. This is a major distinction to this work, where this restriction is absent.

One unit test is described [26] by the execution time it has and by its coverage related to the program or model at hand. The authors used data from the most current version of the software to collect data for test case selection. A traceability matrix was used to connect features to the code. The authors [26] emphasize that this is sometimes not sufficient, as was shown by Ferreira and Barros [82] and needs to be further annotated manually.) The selection is then done by local search, a heuristic approach, or the benchmark of Random Testing.

The article [26] concludes that the heuristic approach produced the best results while providing good results given its lower computational cost.

3.4 Test Case Generation

Test Case Selection and Prioritization can be used in conjunction with Test Case Generation. Here, the techniques can be utilized to reduce the test suite of an automatically generated test suite so that it requires less time for execution. One paper that did this in a novel way was released in 2023 by Rajsingh, Kumar, and Srinivasan [68]. This paper of particular interesting since it proposed a novel Elliptical Distributions-centric Emperor Penguins Colony Algorithm (ED-EPCA) which utilizes the Fishers Yates Shuffled Shepherd Optimization Algorithm (FY-SSOA). The FY-SSOA is one method that can be used for Test Case Selection.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

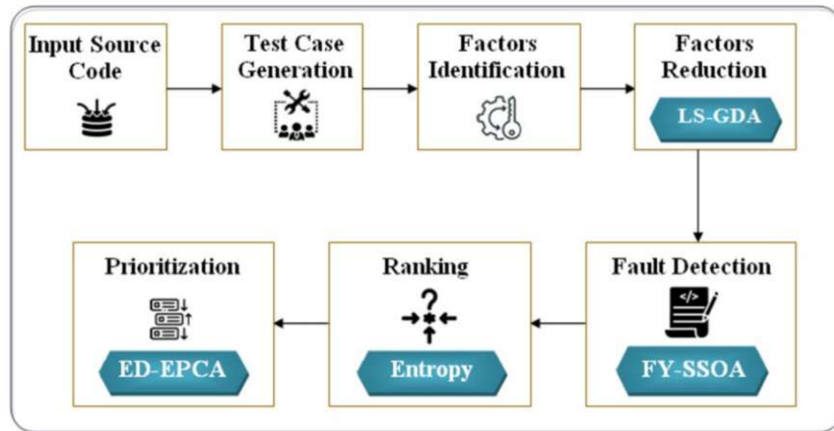


Figure 3.12: Overview of the approach presented by Rajsingh, Kumar, and Srinivasan [68]

The steps in their framework are defined as follows:

1. Input Source Code: This is the source code from a concrete project which shall be tested.

2. Test Case Generation: In this phase, a model is used to generate test cases for the System Under Test.

3. Factors Identification: Because the fault detection rate of the generated test suite may not be 100%, meaning it may not find all bugs present in the implementation, the authors utilized factors for the test case selection. These factors include, among others:

$$Codecoverage = \frac{linesofcodecoveredbytests}{totalnumberoftestcases} * 10$$

$$Dataflow = \frac{testdatautilized}{Totalnumberoftestcases}$$

$$FaultProneness(F_P) = \frac{n_c}{n_p}x10$$

   Where n_c defines the "number of test cases that are correctly pre-dicted as faulty modules" and n_p as the "total number of predicted faulty modules"[68]

4. Factors Reduction: As the name suggests here, the number of factors is reduced by utilizing "Log Scaling-centered Generalized Discriminant Analysis" [68].

5. Fault Detection: The Fishers Yates Shuffled Shepherd Optimization Algorithm is utilized to select test cases. The algorithm was developed to solve optimization problems. The selection step first randomly generates an initial population called a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

community. Then, the fitness value is calculated for every member of the community. Then, the members get shuffled according to the Fisher-Yates algorithm. Then, the step size is calculated, and the position is updated. With this shuffling and the fitness calculation of new populations, the selection is made. The pseudo-code for this algorithm can be seen in Figure 3.11, which was taken from the article [68].

6. Ranking: The selected test cases are now ranked utilizing Entropy. This is defined as follows

$$E^P = -\Sigma S_{tcs} * log(\frac{1}{S_{tcs}})$$

S_{tcs} are denoted as the selected test cases.

7. Prioritization: In the last step, prioritization is done with the Elliptical Distributions-centric Emperor Penguins Colony Algorithm

An overview of the steps of this can be seen in Figure 3.10, presented by the authors in their paper [68].

Pseudocode for proposed FY-SSOA

Input: Reduced factor (R_f)

Output: Selected test cases (S_{tcs})

Begin

Initialize Shepherd population, iteration I_t and maximum Iteration M_t

Initialize position $(R_{f,g})$

Determine fitness value $\rho = N * M$

While $I_t \leq M_t$

    Perform Fisher-Yates Shuffling

    Estimate Step size $S_{f,g} = S_{f,g}^{diverse} + S_{f,g}^{intensify}$

    Determine updated position $R_{f,g}^{new} = R_{f,g} + S_{f,g}$

End while

Obtain (S_{tcs})

End

Figure 3.13: FYSSOA pseudo-code [68]

The selection and prioritization algorithms are especially of interest to this work. The FY-SSOA can be used with optimization problems, which could also be used for this work as the problem can be translated to a knapsack instance.

The presented algorithm usually works very well when compared to alternatives. According to the authors, it produces a higher fitness with the same number of iterations [68].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 4

Methodology

This thesis aims to design a method for time-constraint software testing and evaluate it with a case study. This framework shall allow the selection of test cases of a test suite by inspecting changes pushed to a Source Code Management system. Furthermore, only test cases likely to be affected by changes shall be selected. This means that if a code change affects a test case it shall become a possible candidate for execution. The main differentiation from existing test case prioritization, selection, and minimization techniques is that they shall allow the execution of a suite only if it fits within a provided time budget.

4.1 Research

The design method chapter starts with research, which is required to receive initial ideas on how such a time-constraint software testing method might be constructed. The main focus is literature research into the areas of test case prioritization, selection, and minimization. The goal for this part is to answer the following research question: What characteristics and criteria should be considered in developing a test case selection algorithm that effectively determines the tests within a suite that are susceptible to the impact of pushed commit(s)?

The Qualitative Content Analysis by Mayring [53] is used for this. In his book, Mayring describes various tasks of a qualitative analysis. Relevant are especially the following:

1. Building of Hypotheses and Creation of Theories
2. Classification
3. Checking of Theories and Hypotheses

4.2 Implementation

After researching the state-of-the-art and defining the requirements, the next step is implementing the method. This section aims to answer the following research question: "Which of the determined tests is feasible and essential to execute within a given time constraint to maximize bug detection?"

The design cycle method [86] will be the method of choice. The design and engineering cycle is a process that allows it to work iteratively. The approach defined by Wieringa can be seen in Figure 4.1, taken by the book [86]. Developing this approach is based on prototyping, which is described as a constructive and qualitative method by Wilde and Hess [87].

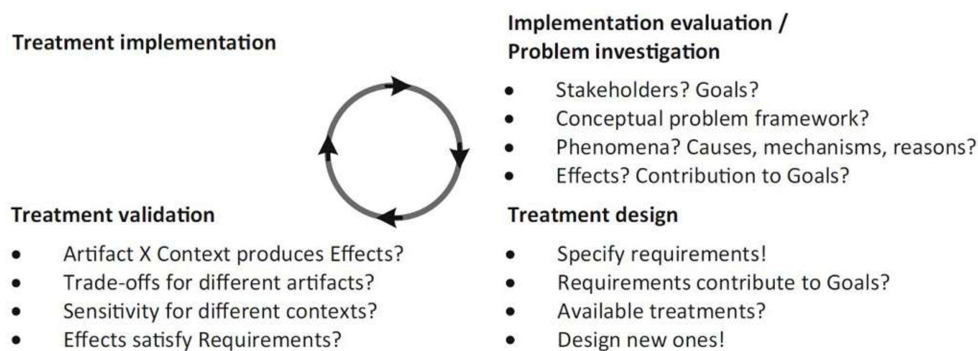


Figure 4.1: Design cycle (Wieringa) [86]

The design cycle [86] consists of the following steps:

1. Problem investigation: As seen in the circle, this phase is about determining what needs improvement and why.

2. Treatment design: In this phase, the focus is on creating at least one design to treat the problem.

3. Treatment validation: This step is used to check whether or not the design treats the problem.

4. Treatment implementation: The design gets implemented to respond to the problem and treat it with the design.

5. Implementation evaluation: At the end/beginning of the circle, the success of the treatment is evaluated, which in turn can trigger a new iteration.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.3 Case Study and Analysis

The method developed in the implementation section shall then be implemented in a company as a case study to see if the approach is feasible. Two research questions shall be answered in this part:

1. How many bugs in the code does such an approach find compared to running the whole test suite?
2. What is the relationship between the duration of the time budget and the effectiveness of a selected test set, and how can the optimal time budget be determined for achieving the best results?

Different iterations will be needed to deploy and fit the method onto a project. Johansson [41] describes a case study as follows:

> A case study is expected to capture the complexity of a single case, which should be a functioning unit, be investigated in its natural context with a multitude of methods, and be contemporary. A case study and, normally, history focus on one case, but simultaneously take account of the context, and so encompass many variables and qualities [41].

Wilde and Hess see the case study as a behaviouristic and qualitative method [87].

4.4 Research Questions

To recap all the research questions from chapter 1.3:

1. RQ1: What characteristics and criteria should be considered in developing a test case selection algorithm that effectively determines the tests within a suite that are susceptible to the impact of pushed commit(s)?
2. RQ2: Which of the determined tests are feasible and essential to execute within a given time constraint in order to maximize bug detection?
3. RQ3: How many bugs in the code does such an approach find compared to running the whole test suite?
4. RQ4: What is the relationship between the duration of the time budget and the effectiveness of a selected test set, and how can the optimal time budget be determined for achieving the best results?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Method Design

This chapter focuses on developing a prototype for a time-constraint testing approach so that the feasibility can be tested. As the evaluation will be done with a case study project, the goal during development is to work in an iterative fashion and test with the case study project as early as possible. The reason for doing this is to test the approach on a real project with a real scope as easily as possible and not use a small and/or artificial project.

Generally, the prototype will be implemented as a framework and is not project-specific. More detail on this is presented in the requirement analysis (chapter 5.2). The development will be done in iterations with the goal to develop an Minimum Viable Product (MVP) first, then test it on the project and improve possible shortcomings. For each iteration (except for iteration 0, which is the requirement analysis), the design cycle method will be used (described in chapter 4.2) where each iteration consists of the following parts:

1. Implementation Evaluation and Problem Investigation: At first, the implementation of the previous iteration is evaluated. This, for example, means in Iteration 6, the code base of Iteration 5 is evaluated. The same five artificially constructed bugs are used. These artificially introduced bugs are used for repeatability. The Implementation Evaluation is not always required and is, in some iterations, omitted.

   The Problem Investigation updates the requirements based on findings from the previous iteration and the evaluation of the previous chapter.

2. Treatment Design: This section is focused on creating or updating the implementation design so that the updated requirements are addressed. In some iterations, only a single design is made, and sometimes it is more than one.

3. Treatment Validation: In this section, the design is analyzed to check if it can address the requirements, and if not, the design is adjusted.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4. Treatment Implementation: Here the design is implemented, and then the next iteration starts again with the evaluation.

5.1 Iteration Overview

Below there is a short description of the different iterations:

0. Iteration - Requirement Analysis: The discussion of the requirements for the implementation.
1. Iteration: First research possible approaches on how a time-constraint testing framework could be implemented. The decision was made to test a backtracking approach. Therefore, a backtracking approach was designed and implemented.
2. Iteration: The evaluation showed that the backtracking approach is not feasible, and the decision was made to switch to a genetic algorithm instead.
3. Iteration: The goal of this iteration was to deploy the approach to the CI system of the case study project with the unit tests.
4. Iteration: Here the goal was to deploy the current state to the case study projects UI tests as they take more time than the unit tests (90 minutes).
5. Iteration: The approach was improved to keep the time budget better. (As the budget is not strict - check requirements (chapter 5.2)
6. Iteration: The Random Testing (RT) approach was developed as a control group for the developed method (Iteration 6).
7. Iteration: The RT approach was improved to better adhere to the time budget.
8. Iteration: The RT was evaluated and the decision was made to continue with the Evaluation and Results (chapter 6).

5.2 0. Iteration - Requirement Analysis

The requirements of a time-constrained software testing approach are presented below and are the basis for the iterations presented in this chapter. They can be derived from the idea of such an approach and the data that is required to execute it.

1. Test Case Selection: The main goal is to select test cases that are most likely to fail. It is left to the implementation phase to determine how to achieve this.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

2. Time Boxing/Budget: Providing a time budget to the system and adhering to the budget are some of the most basic functions. One key decision is whether the time budget should be strict. A strict time budget forces the execution to be forcefully stopped after a predetermined period of time. The decision is that the time budget shall not be strict as otherwise a custom test runner would be needed. Rather, the program should do its best to adhere to the time budget but not enforce it. It is acceptable if the test execution is finished.

3. Running on Continuous Integration (CI): The goal is for the approach to have a particularly high ease of use. It shall be useable in a CI pipeline.

4. Nightly Data Collection: As some information will be needed to decide which test cases will likely fail, this information will be collected during a nightly job and stored in a database. A fitting database needs to be selected.

5. Transferable: The framework must be implemented to make it transferable to other projects. The concrete implementation can require a specific stack like Java with a specific test/coverage framework. However, the techniques must be transferable.

5.3 1. Iteration

5.3.1 Problem Investigation

The problems/goals for the first iteration:

1. Conduct literature research and investigate possible approaches.
2. Select a fitting approach.
3. Select a technology stack.
4. Implement a method to find tests affected by code changes.

Research

The goal for this iteration is first to select an approach. In order to do this, the first thing required is to conduct literature research. The goal is to get an overview of existing test case prioritization, selection, and minimization techniques. This is driven by RQ1: "What characteristics and criteria should be considered in developing a test case selection algorithm that effectively determines the tests within a suite that are susceptible to the impact of pushed commit(s)?"

Articles found during the literature review are described in the third chapter, "related work". The approaches used in articles, whether test case prioritization, selection, or minimization, fall into one of two categories. They either use heuristics like Additional or Total Coverage [58] [76] or sophisticated techniques including approaches like genetic algorithms [26] and Coverage-Based Ant Colony System [51].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Especially the approach that Farzat and de O. Barros used [26] yields good results and is easier to implement than other approaches like the CB-ACS approach [51].

One discovery made is that Jest [1], the JavaScript testing framework, supports '–changedOnly'. An investigation into the implementation revealed that backtracking is utilized. The developers used a framework to retrieve changes from Git. The imports are analyzed to get the files from which the code was imported. The imports are then repeatedly traced until a test file is found. With this method, they found the test files that directly or indirectly imported a changed file. Therefore, these test files might cover the code that was changed.

The initial idea for the implementation is to do the same thing: first, backtrack and then improve the results from there. The goal is to execute all the test files found through backtracking and then run an evolutionary algorithm to cover the remaining files. The framework for the approach is developed in NodeJS. The first iteration simply shall design and implement the backtracking.

This resulted in the following requirements:

1. It should be out of the box applicable to NodeJS projects.
2. It should read the Git[2] history and find files changed since the fork of a new branch or a configurable starting point.
3. It should implement backtracking to find files affected by code changes.

5.3.2 Treatment Design

To retrieve changes files the project utilizes the 'jest-changed-files' [3] Node Package Manager (NPM) package. The implementation takes the same approach as Jest for the backtracking, where it considers the file imports. It was considered to trace the more fine-grained function calls. However, associating the function calls and the source code of the functions, then tracing the usages and doing that until a test file is found, is more complex. The approach traces the imports to the file and then the imports of those files back to the files until you reach a test file.

Figure 5.1 shows the design for the backtracking algorithm and the supporting files. The program is implemented in a modular fashion. The goal of the different classes is as follows:

- Main: The entry point of the implementation. This is called the testfiles module, which gathers the tests to execute and then executes them.

[1] https://jest-archive-august-2023.netlify.app/docs/28.x/cli#–onlychanged, accessed on: 05. April 2024

[2] https://git-scm.com/, accessed on: 05.April.2024

[3] https://www.npmjs.com/package/jest-changed-files, accessed on: 05.04.2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Testfiles: This serves as the main implementation for the backtracking. The file orchestrates the reading of the test files from the file system. Furthermore, this reads the Git history and compares the state of the paths provided against a specified commit or branch. The commit to be compared against can be specified. Then, it will backtrack the files read and return the test files to execute.
- ReadDirectoryHelper: Provides the file structure from the file system. Having this separate from test files allows changing frameworks quickly if needed.
- TypeHelper: This is an implementation to check the file types of the files read.

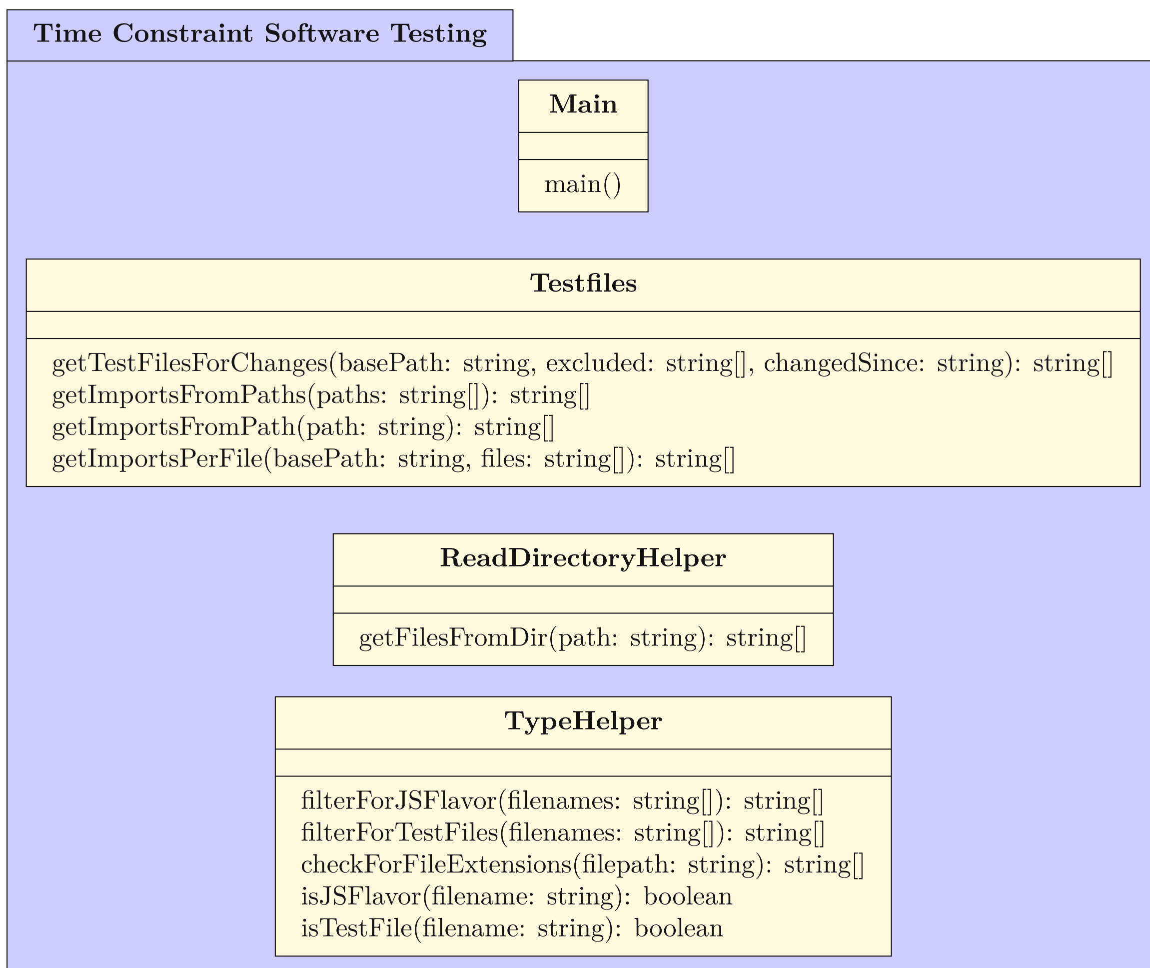


Figure 5.1: Iteration 1: Backtracking architecture design

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.3.3 Treatment Validation

The main focus for validating the approach is to check if backtracking finds introduced bugs. To ensure this, the functionality offered by Jest is investigated. Their implementation did find the bugs. Since there was no blocker, the implementation of the approach was started.

5.3.4 Treatment implementation

The implementation of the class diagram is simple. Complex is the backtracking algorithm. The code is presented in algorithm 5.1. It consists of the following steps:

1. Read all source files from the file system.
2. Create a map that associates which filename imports which files.
3. Read which files were changed.
4. Read the test files.
5. Backtracking for each changed file:
    a) If the changed file is a test file, add it to the result.
    b) Iterate over all test files:
        i. Create an empty, visited array.
        ii. Create a toExplore array with the test file in it.
        iii. As long as toExplore is not empty for each element of toExplore:
            A. Take one element to explore.
            B. Add the element to visited.
            C. Add all imports of the element that were not visited into a temporary list.
            D. If the temporary list includes a changed file we can add the current test file to the result and break out of the loop.
            E. Else, add the elements to be Explored and continue.
6. Return results.

This algorithm starts from the test files and backtracks through the imported files. If a file is reached at any point, it is selected. The only optimization implemented is the abort. The program does not backtrack further as soon as a test is found. If it backtracks to a changed file, the file is selected. The visited list ensures the program does not enter an endless loop due to of circular file imports.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.1: Iteration 1: Backtracking algorithm implementation

Data: basePath, excluded, changedSince

Result: Array test names

```
1  const files = await glob(basePath + '/**/*',  ignore: excluded.map((x) =>
     `basePathx/**`) );
2  const importPerFile = await getImportsPerFile(basePath, files);
3  const changedFiles = Array.from( ( await getChangedFilesForRoots([basePath],
     changedSince, ) ).changedFiles, );
4  const testFiles = filterForTestFiles(files).map((x) => resolve(`$x`));
5  const resolveTestFilesToChangedFiles = () =>
6  {
7     const fileTypesToIgnore = ['.spec.ts.snap', '.spec.tsx.snap'];
8     const testFileForChangedFile = new Map<string, string[]>();
9     for (const changedFile of changedFiles)
10    {
11       if (fileTypesToIgnore.some((x) => changedFile.endsWith(x)))
12       {
13          continue;
14       }
15       if (isTestFile(changedFile))
16       {
17          testFileForChangedFile.set(changedFile, [changedFile]); continue;
18       }
19       const foundTestFiles: string[] = [];
20       for (const testFile of testFiles)
21       {
22          const visited: string[] = [];
23          const toExplore = [testFile];
24          while (toExplore.length > 0)
25          {
26             const element = toExplore.pop();
27             if (element)
28             {
29                visited.push(element);
30                const imports = importPerFile.get(element);
31                const toAddToExplore = imports?.filter((x) =>
                     !visited.includes(x)) ?? [];
32                if (toAddToExplore.includes(changedFile))
33                {
34                   foundTestFiles.push(testFile); break;
35                }
36                toExplore.push(...toAddToExplore);
37             }
38          }
39       }
40       testFileForChangedFile.set(changedFile, foundTestFiles);
41    }
42    return testFileForChangedFile;
43 }
44 ;
45 return resolveTestFilesToChangedFiles();
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.4 2. Iteration

5.4.1 Implementation Evaluation/ Problem Investigation

The backtracking approach was tested with the unit tests of the case study project. Five artificial bugs were introduced into the case study project for the evaluation. Then, the backtracking algorithm was executed, and how long each execution took and how many bugs were found were noted.

The execution was run with one, two, three, and five manually introduced bugs. For each of these bugs, the pipeline was executed ten times, for a total of 40 executions.

The executions with one and two bugs found all introduced bugs (one and two, respectively). The third bug was not detected by any run, and the executions with five bugs found all four bugs except the third.

	1 Bug	2 Bugs	3 Bugs	5 Bugs
1.	2m 29s	2m 39s	2m 43s	5m 24s
2.	2m 40s	2m 40s	2m 47s	5m 21s
3.	2m 42s	2m 45s	2m 45s	5m 26s
4.	2m 26s	3m 01s	2m 32s	5m 20s
5.	2m 28s	2m 40s	2m 39s	5m 56s
6.	2m 47s	2m 48s	2m 46s	5m 36s
7.	2m 27s	3m 07s	2m 42s	6m 14s
8.	2m 41s	2m 58s	2m 49s	5m 23s
9.	2m 28s	3m 15s	2m 44s	5m 58s
10.	2m 45s	2m 39s	2m 46s	5m 55s

Table 5.1: Evaluation of iteration 1: Execution times of backtracking run

The backtracking approach provided a substantial portion of the suites tests. The main issue is that for the execution with five bugs, which just had the bugs in the code and no other files were changed, implying that this is a best-case scenario, a time budget of below 50 percent can not be kept.

Information about the execution times can be seen in Figure 5.2, which shows a box plot of the execution durations. There is almost no timing difference between the pipelines with one and two bugs. A detailed investigation into the test cases selected showed that the reason for this is that there were not a lot of tests covering the second bug. There is no timing difference between two and three bugs (except for the performance difference of the virtual machines). This is because there were no differences in the number of tests selected. However, the increase in the duration of five bugs is notable. The pipeline selects over 50 percent of all tests available. While it would be possible to reduce the tests selected, it was decided to discard the approach and investigate alternatives.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

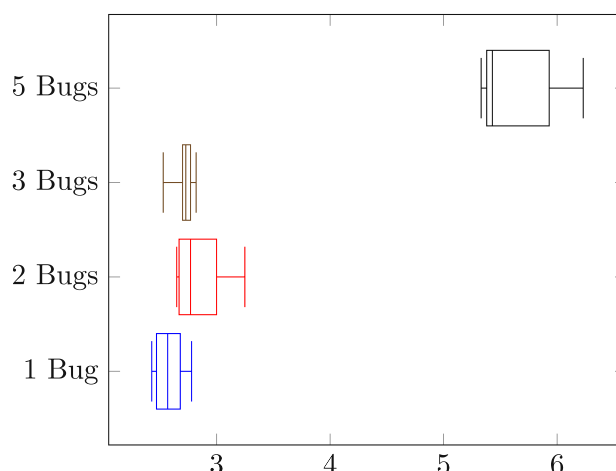


Figure 5.2: Evaluation of iteration 1: Boxplot backtracking execution times from table 5.1

The next goal is to switch the design to a different approach that can keep the time budget. As expressed in iteration 1, two options are available for this different approach. A heuristic such as Additional or Total Coverage [58] [76] or a more sophisticated technique such as genetic algorithms [26] and Coverage-Based Ant Colony System [51]. The genetic algorithm approach is chosen for the iteration because of the reduced complexity compared to Coverage-Based Ant Colony System and the results shown by Farzat and de O. Barros [26]. Like this work, the goal of Farzat and de O. Barros in their article was to reduce a test suite to a time budget. However, the authors restricted their approach to unit tests and did not consider new features. These restrictions are not made in this work.

The needed information for the new approach is stored in a database. This data is collectible in a nightly test execution. For this iteration, it is sufficient to execute everything locally. Test case information has to be stored on a per-test file basis. This reduces complexity and can be adjusted to be more fine-grained later.

For this iteration, the following additional requirements were specified:

1. It should be possible to collect data about test cases.
   a) It should be possible to execute test case data storage in a nightly pipeline and store the data in a database.
   b) It should store the execution time of test cases on a test file basis.
   c) It should collect and store coverage information of test cases on a test file basis.
2. It should be possible to execute test cases to fit within a specified time budget.
3. It should optimize test cases with a simple genetic algorithm.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.4.2 Treatment Design

The design aims to create a basic genetic algorithm; it must be expandable in future iterations. The fitness function has to be easily swappable, with the goal of maximizing total coverage.

The idea for the genetic algorithm's fitness is to calculate two fitness values, one representing the percentage of total coverage and the other representing the percentage of coverage of the changed files. A simple fitness value is calculated by multiplying each value with a tuning parameter and then summing them up. Files affected by changes have a bigger impact than the others.

For the design, the first challenge is the collection of coverage information from test files. Mocha was selected as a testing framework. It is difficult to collect the coverage data per test file because Mocha counts how often a given statement, function, or branch is covered. So, to get the coverage per test file, it is necessary to write a custom reporter that stores the coverage after each file change and subtracts the coverage of the previous file. This data has to be written into a Comma-Separated Values (CSV) file as Mocha does not allow asynchronous calls in its reporters, meaning it cannot be persisted in the database directly, but writing the data into a file can be done synchronously.

Afterward, a script iterates over the exported coverage data and persists it into a Neo4j database. The Neo4j database is used because it is geared toward queries representing paths, which is useful when retrieving the data. The execution time of each test file is persisted to study timing differences. The time limit in the genetic algorithm is realized with the fitness function. It returns a fitness of 0 if the proposed test set exceeds the time limit.

Program Design

Figures 5.3 to 5.5 show the proposed design of the simple genetic algorithm. Compared to the first iteration, the design is done in an Object Oriented Programming (OOP) fashion. The decision to move to OOP was made to change classes and optimize the algorithm in future iterations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The classes do the following:

- TestfileAdapter: Used to read the file structure and the changed files from the file system. Some details here can be taken from Iteration 1.
- TestExecutor: This class is responsible for executing testfiles. The source directory in which the execution command shall be run can be specified, as can the base command to be used. This should allow me to change from Mocha to another testing framework in future iterations or employ a different coverage framework.
- Neo4JAdapter: The idea behind this class was to make the database easily changeable. At the same time, the investigation suggested that Neo4J would be a good fit. It was unclear at the time of the decision if everything could be implemented satisfyingly.
- Genome: This class was responsible for storing which test cases are selected and allowing for things like mutation. This is a more sophisticated data structure used for storage.
- Population: A population was one evolution within the genetic algorithm, an abstraction of the genome with additional functionality.
- Fitness: An interface for calculating the fitness value of a population.
- BasicFitness: An implementation of the Fitness interface using total coverage in a boolean fashion for each line.
- Genetic: The implementation of the genetic algorithm itself, supporting things like evolution and setting of tuning parameters.
- File: An abstraction of a source code file.
- Test: An abstraction of a test file.
- Covers: An abstraction for the coverage relation between tests and files, also storing which statement is covered.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

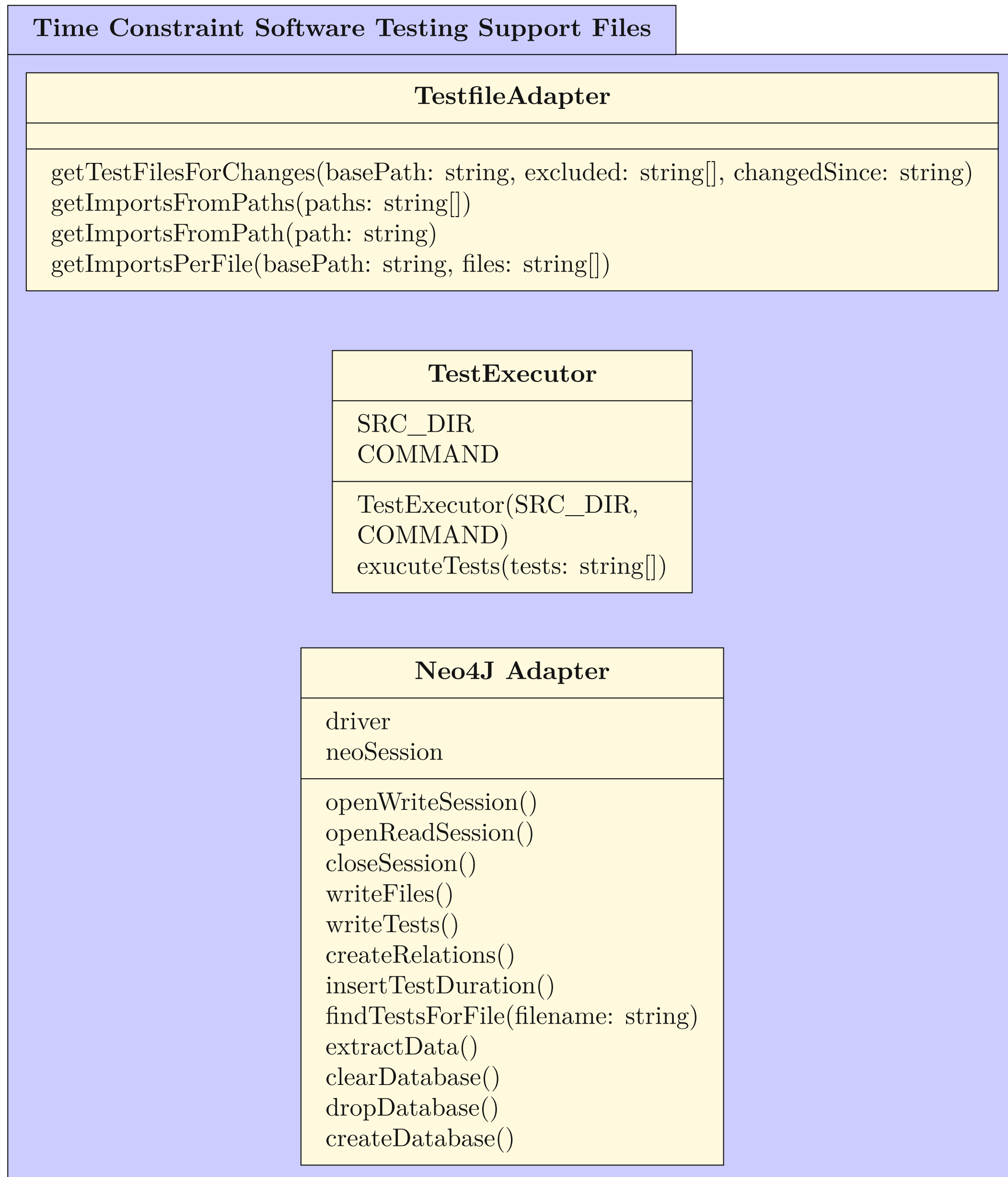


Figure 5.3: Iteration 2: Genetic algorithm architecture design

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Figure 5.4: Iteration 2: Genetic algorithm architecture design

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

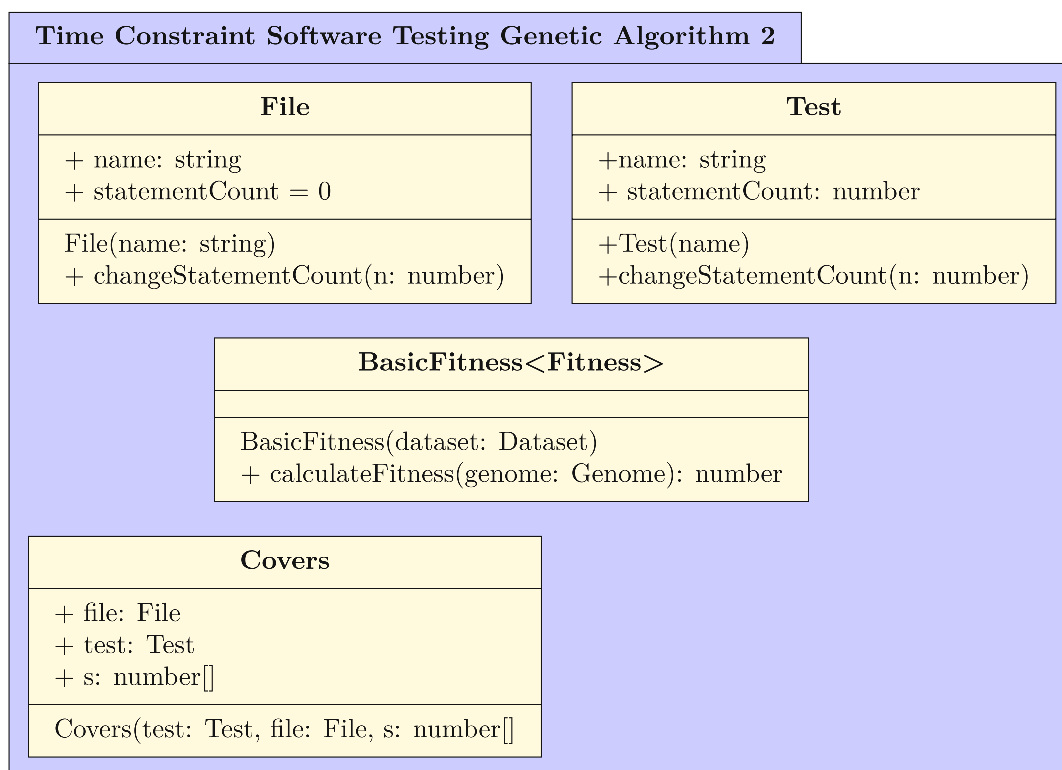


Figure 5.5: Iteration 2: Genetic algorithm architecture design

5.4.3 Treatment Validation

The design was evaluated, and the time budget for the fitness was added. The fitness implementation was given special consideration during the evaluation as it was unclear if it would fulfill the requirements, notably an ascending value for increased coverage. The implementation can be seen in Algorithm 5.2.

The fitness is calculated twice to consider both files changed and the general coverage of the whole project. One of these fitness values considers only changed files, while the second considers all values. The one with the changed values is then multiplied with a tuning parameter to emphasize the importance of the changed files. For the general value, both are summed up.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.2: Initial basic fitness

Data: Genome
Result: Fitness

```
1  const coveragePerFile = new Map<string, number[]>();
2  const testFileCovers = this.dataset.testFileCovers;
3  const includedTests = this.dataset.retrieveTests(genome);
4  let duration = 0;
5  for (const test of includedTests)
6  {
7      duration += test.duration;
8      for (const filename of testFileCovers.get(test.name)?.keys() ?? [])
9      {
10         const coverage = testFileCovers.get(test.name)?.get(filename);
11         if (coveragePerFile.has(filename))
12         {
13             const newCoverage =
                 combineStatementCoverage(coveragePerFile.get(filename) ?? [],
                 coverage?.s ?? []);
14             coveragePerFile.set(filename, newCoverage);
15         }
16         else
17         {
18             coveragePerFile.set(filename, coverage?.s ?? []);
19         }
20     }
21 }
22 if (duration > this.timeLimit)
23 {
24     return 0;
25 }
26 let numberOfLinesCovered = 0;
27 for (const filename of coveragePerFile.keys())
28 {
29     const coverages = coveragePerFile.get(filename) ?? [];
30     for (const coverage of coverages)
31     {
32         numberOfLinesCovered += coverage > 0 ? 1 : 0;
33     }
34 }
35 return numberOfLinesCovered / this.dataset.totalNumberOfLinesToTest;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.4.4 Treatment Implementation

The implementation phase took some time due to the complexity and the need for the components to work together. The script is a crucial component, as it works as an orchestrator for the genetic algorithm and is, for example, responsible for setting tuning parameters.

In Algorithm 5.3, the source code for this script can be seen. It consists of the following steps:

1. Setup necessary data structures like tests, dataset, fitness, and the genetic algorithm.
2. Calculate the tuning parameters and set them.
3. Run the genetic algorithm.
4. Retrieve the best population and the best test set from it.
5. Execute best test set.

Algorithm 5.3: Genetic algorithm orchestration script

Data: ROOT_DIR, SRC_DIR, TEST_DIR, CHANGED_SINCE
Result: Array of issue titels

```
1  // setup necessary files
2  const tests = getChangedTestFiles();
3  const dataset = new Dataset(...);
4  const fitness = new BasicFitness(dataset, timeBudget);
5  const genetic = new Genetic(dataset, fitness);
6  const avgDuration = tests.map((t) => t.duration / tests.length).reduce((a, b)
     => a + b);
7  let prob = timeBudget / avgDuration / tests.length;
8  genetic.setTuningParameters(0.05, 200, 10, 10, prob > 1 || prob < 0 ? 0.5 : prob);
9  // run genetic algorithm
10 for (let i = 0; i < 100; i++)
11 {
12     genetic.evolve();
13 }
14 bestPopulation = genetic.getBetPopulation();
15 bestTestSet = genetic.getBestTestSet();
16 // execute tests
17 executeTests( ROOT_DIR, options.command, bestTestSet.map((t) =>
     resolveToChildDirectory(t.name)), );
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.5 3. Iteration

5.5.1 Implementation Evaluation/ Problem Investigation

To evaluate the genetic algorithm, the approach is tested on a local machine (a Lenovo P1 6th gen with an Intel i9 processor and 64 GB of memory). The algorithm was run in two variations. The first one used 100 evolutions and a population of 50. The second one had 500 evolutions and a population size of 100. The same 5 bugs were used for the evaluation as for the backtracking evaluation.

Table 5.2 shows the amount of time needed for a time budget of one, two, three, and five minutes. Two things stand out from the collected results. The first one is that the actual time varied quite drastically. For example, for the one-minute proposed time budget, the lowest observed time was 15 seconds, and the highest was almost two minutes at 1m 52s. This is more than seven times as much as the lowest value. The same can be observed with the five-minute time budget: the lowest was 44 seconds, and the highest was over twelve minutes. This huge variance between runs is immediately apparent in the boxplot in Figure 5.6.

	1 min	2 min	3 min	5 min
1.	1m 09s	0m 26s	4m 07s	0m 44s
2.	0m 15s	0m 40s	0m 38s	6m 19s
3.	0m 17s	0m 27s	3m 56s	1m 24s
4.	1m 52s	0m 44s	0m 23s	12m 19s
5.	0m 46s	0m 29s	3m 50s	11m 16s
6.	0m 20s	0m 26s	3m 59s	11m 58s
7.	0m 29s	0m 30s	0m 33s	10m 57s
8.	0m 15s	0m 34s	3m 56s	11m 12s
9.	0m 19s	0m 26s	0m 31s	1m 28s
10.	0m 35s	0m 28s	0m 26s	11m 02s

Table 5.2: Evaluation of iteration 2: Execution times of different time budgets and 5 bugs introduced. (100 iterations, population: 50)

Table 5.3 shows the number of bugs each pipeline found. One thing that stands out is that except for four instances, no bugs or all bugs were found (in the four exceptions, one bug was found). This justified a deeper investigation. The investigation concluded that the calculation of the fitness function caused this behavior, especially the fitness of changed files, as the number of changed files is low in comparison to a real commit. The fitness value did not increase as sharply as needed always to include those files.

The results show that the approach can work. The assumption is that the algorithm would yield better results with more iterations and a bigger population. If that is the case, it would indicate that more tuning is needed. To test this, the approach was executed with 500 iterations and a population size of 100.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

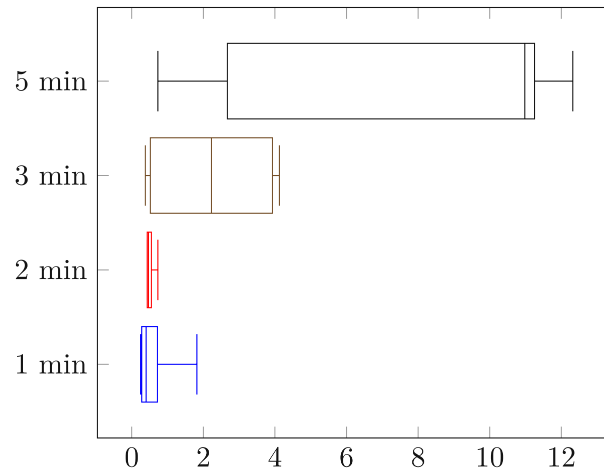


Figure 5.6: Evaluation of iteration 2: Boxplot of execution times from table 5.2

	1 min	2 min	3 min	5 min
1.	0	0	5	0
2.	0	0	0	5
3.	0	0	5	1
4.	5	0	0	5
5.	0	0	5	5
6.	0	1	5	5
7.	0	0	0	5
8.	0	0	5	5
9.	0	0	0	1
10.	0	0	0	5

Table 5.3: Evaluation of iteration 2: Number of bugs found with different time budgets and five bugs introduced. (100 Iterations, Population: 50)

Table 5.4 and Figure 5.7 show better adherence to the time budget. The same improvement is present for the number of bugs found, as seen in Table 5.5. However, the results did not meet the requirements. The conclusion is that improvements are required.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

	1 min	2 min	3 min	5 min
1.	2m 14s	1m 54s	2m 49s	10m 11s
2.	3m 02s	4m 43s	5m 57s	8m 42s
3.	2m 48s	4m 45s	5m 58s	8m 32s
4.	5m 38s	4m 31s	6m 09s	8m 17s
5.	2m 49s	2m 01s	2m 43s	9m 33s
6.	0m 46s	1m 57s	2m 56s	9m 14s
7.	2m 56s	1m 50s	7m 11s	8m 47s
8.	2m 50s	4m 40s	3m 03s	9m 03s
9.	1m 17s	2m 05s	2m 54s	8m 14s
10.	4m 16s	4m 35s	7m 26s	10m 04s

Table 5.4: Evaluation of iteration 2: Execution times of different time budgets and five bugs introduced (500 iterations, population: 100)

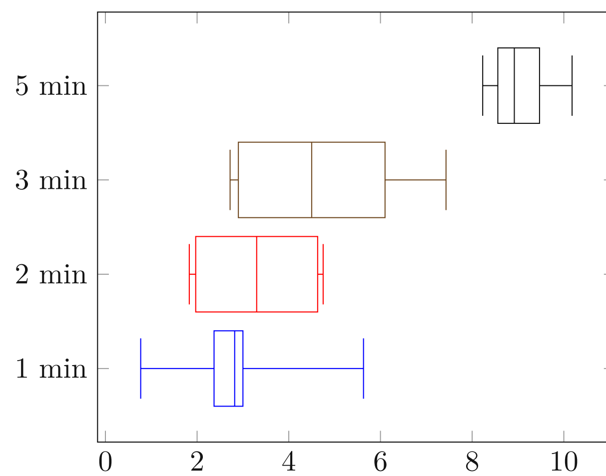


Figure 5.7: Evaluation of iteration 2: Boxplot of execution times from table 5.4

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

	1 min	2 min	3 min	5 min
1.	0	0	0	5
2.	0	5	5	5
3.	0	5	5	5
4.	5	5	5	5
5.	0	0	0	5
6.	0	0	0	5
7.	5	0	5	5
8.	5	5	0	5
9.	0	0	0	5
10.	5	5	5	5

Table 5.5: Evaluation of iteration 2: Number of bugs found with different time budgets and five bugs introduced. (500 Iterations, Population: 100)

A few goals and requirements are open from Iteration 1, and new ones were added:

- It should stick better to the time budget.
- It should find more failing tests in a shorter time.
- It should run on the CI system.
- It should use a database system that does not require a pro version.

The issue with the algorithm is the fitness and the selection. This will be addressed at a later point. This iteration aims to make the systems work on the Continuous Integration and deploy it to the case study project. This makes repeatable testing simpler and results comparable to the numbers expected at the end of the project. One problem is that it takes time for Neo4J to clear the database. This was mitigated by dropping the database in Iteration 2. However, this was unfeasible in the CI because dropping a database requires supporting multiple databases, an enterprise feature. Thus, it was decided to switch to another database system.

5.5.2 Treatment Design

As the system went into trial deployment for the case study project, a branch of the project was created. Additional pipeline steps with the time budget are added. A nightly job is scheduled for this branch to set up and update the database's coverage data. The nightly pipeline does this by executing all tests and overwriting the coverage data.

The approach was enhanced by automatic data collection. For each pipeline, it is stored which test files were selected and which failed, the duration of each test, the time budget specified, and the time it took to execute the pipeline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

PostgreSQL was used as the database system for ease of use. The schema for the new database is presented in Figure 5.8. Notably, a “Run” and a “Subpipeline” table are added. The “Test” and “File” tables, and the “covers” relation are translations of the types and relations in Neo4j. The new adapter, Figure 5.9, shows the additional functions required for data collection.

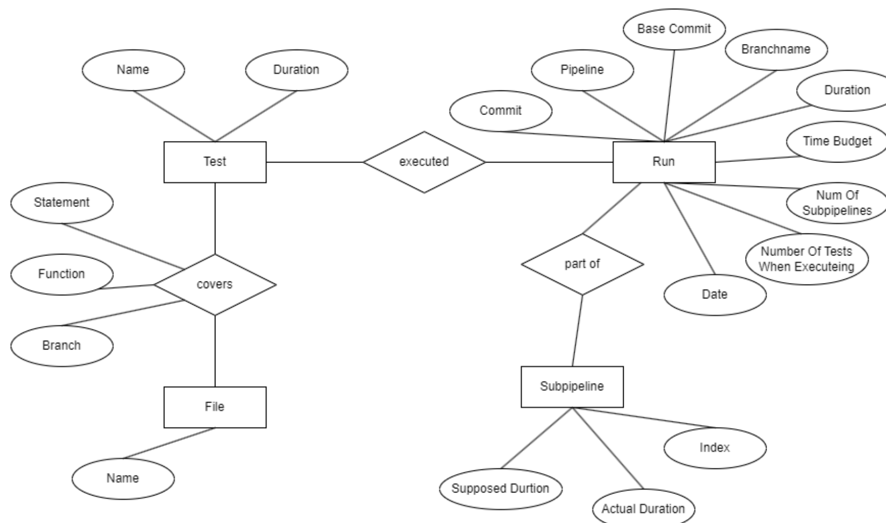


Figure 5.8: Iteration 3: Entity relation diagram of database

5.5.3 Treatment Validation

Again, like in the previous iteration, the algorithm did not stick to the time budget, and it did not find sufficient bugs. Furthermore, it is clear from the statistics that the setup and tear-down process of the test suite is impacting the number of tests that can be run. The issue is that the setup process requires different amounts of time because of variations in hardware and the utilization of virtual machines. The decision is not to consider this when calculating the time budget for better comparability and just incorporate the actual time required for test execution.

5.5.4 Treatment implementation

The prototype from the design phase was used and improved. Disposing of setup and tear-down time was implemented, and time budgeting was adjusted to accommodate these changes.

One issue that occurred in conjunction with GitLab, which is the CI system the project is hosted on, is that when GitLab fetches a pipeline, it initiates the History with an empty git repository. Therefore, to get the history, it was necessary to manually fetch and pull the main branch. Due to a limitation of GitLab, the current branch can only fetch the last 1000 commits at most. This is a non-issue as it can be avoided if the cloning is done

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

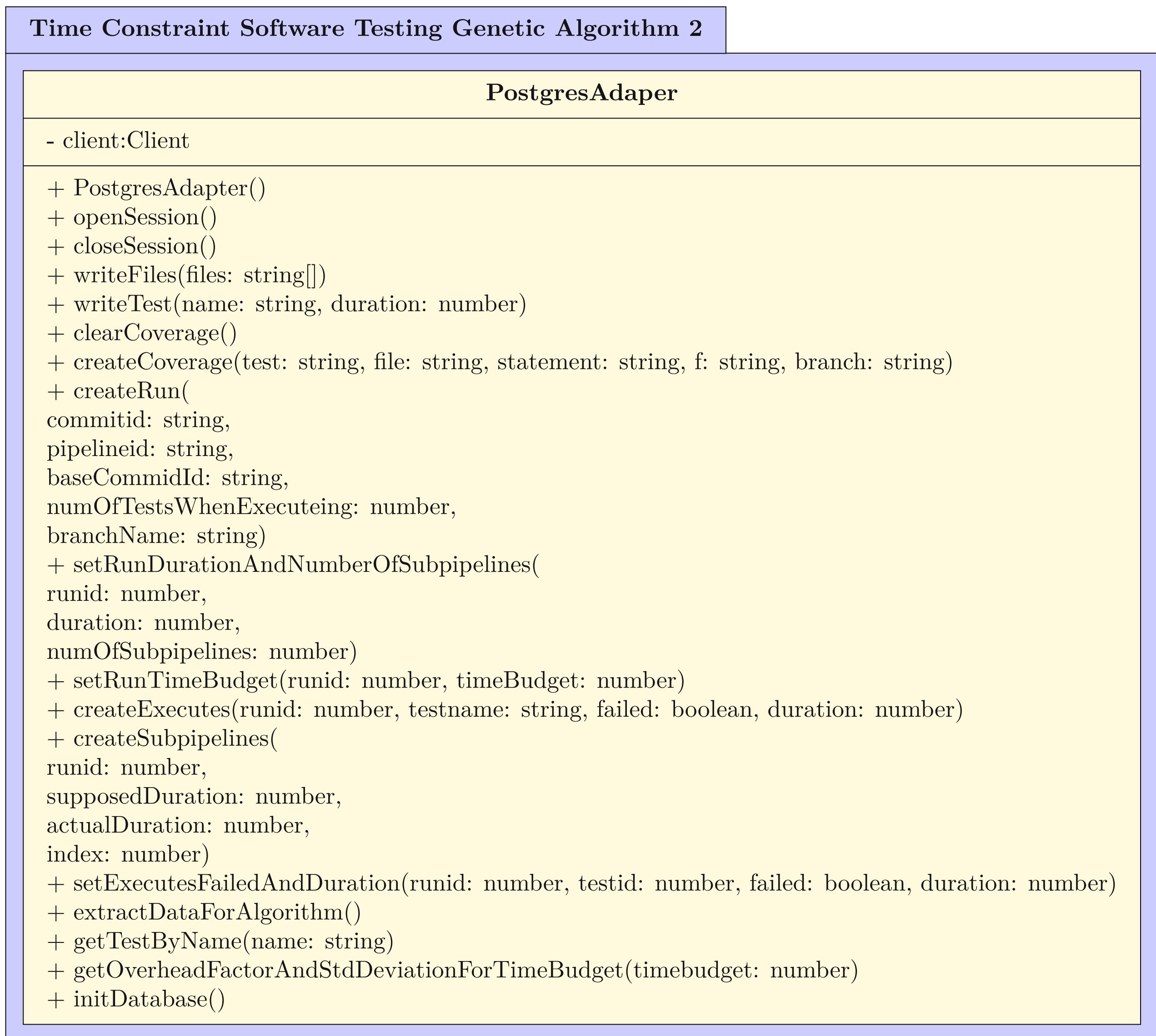


Figure 5.9: Iteration 3: Postgres adapter design

manually. It was tried to make an automated design for this. However, after trying to figure out how to construct this, it was decided to continue with the implementation and leave this for a possible future project.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

5.6 4. Iteration

5.6.1 Implementation Evaluation/Problem Investigation

At this point it was observable that the stack of the implementation is functional. However, a goal is to get it running on the User Interface (UI) tests. This is because this testing section takes two and a half hours. The idea is to test the performance of the approach on long pipelines. Therefore, with the previous unfulfilled requirements, these are the currently open requirements:

- It should stick better to the time budget.
- It should find more failing tests in a shorter time.
- It should run with the UI tests

5.6.2 Treatment Design

As the goal is to get the approach running on UI tests, it has to be noted that for the case study project, these serve as integration tests if these tests do not require a physical device. The software usually needs to run on a purpose-built device with specialized hardware. The first problem that was found was that the pipeline that ran the UI tests utilizes psexec [4] to run the tests on Windows and get an image.

This is necessary to provide environment variables for the application over psexec. The command length limitation of psexec in combination with Windows CMD became the issue. While this limit is usually at 8191 characters [5], it was much shorter. So much so that it became an issue that was hard to spot. The solution is to write the commands into a ps1 script on the executing virtual machine during pipeline execution and execute that. Another issue could be observed: for some reason, the propagation of the error code was not working as intended, meaning it was not forwarded to the parent program, resulting in pipelines that should have failed but showed up green. This bug was fixed by reading the exit code of the child process.

5.6.3 Treatment Validation and Implementation

For the validation, one obstacle became apparent. Collecting coverage information with WebdriverIO [6] is only possible if using it to do component testing and not desktop testing, which is what the project is using. This is not straightforward to rewrite and would require a lot of effort.

[4]https://learn.microsoft.com/en-us/sysinternals/downloads/psexec, accessed on 07.02.2024

[5]https://learn.microsoft.com/en-us/troubleshoot/windows-client/shell-experience/command-line-string-limitation, accessed on 07.02.2024

[6]https://webdriver.io/docs/component-testing, accessed on 07.02.2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The other design option would be to completely rewrite either the stack of the case study or write a custom test case executor. As the goal was to test the approach's feasibility, it was decided to skip testing with the project's UI tests and focus on the unit tests. While the unit tests do not run as long, they still provide interesting data for analysis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.7 5. Iteration

5.7.1 Implementation Evaluation/Problem Investigation

This iteration aims to address the two requirements open from the last iteration. No new requirements were added.

- It should stick better to the time budget.
- It should find more failing tests in a shorter time.

5.7.2 Treatment Design

During the preparation of the implementation evaluation, the best results of the current implementation were investigated. Different parameters for tuning the evolutionary algorithm were tested. The algorithm struggles to find test suites within the specified time frame. This is caused by two things: The first issue is that considering tests with coverage of general code and changed code simultaneously results in a poor test selection. As stated in Iteration 2, the tests affected by changes are low in number. Therefore, the fitness of these tests needs to be increased significantly so that shorter tests with higher coverage of the general area are not preferred. The second problem is that the prediction of the duration of the tests is inaccurate.

Possible solutions are investigated to address the issue and understand how the requirements can be better addressed.

The first plan is to use statistics from the previous pipeline run to adjust the time budget so that it is longer or shorter according to the average overshoot. For example, suppose a pipeline with a time budget of 60 seconds usually takes 80 seconds. It is possible to calculate the percentage of the time budget that is exceeded and then reduce the original time budget by that percentage. This is easy to implement thanks to the data collection for the statistics.

The other idea is to split the pipeline into two sub-pipelines. First, the algorithm considers tests associated with changes with higher probability (giving it a higher tuning factor than before). Moreover, the first run collected how big the difference was between the time observed and the time that was calculated in advance. This value is then used for the second pipeline to fill the time budget. The second run checks if there are still tests associated with changes remaining. If so, the calculation will stay the same. If not, it will create the best coverage for the whole project if there is still time. This approach can be seen in Figure 5.10.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

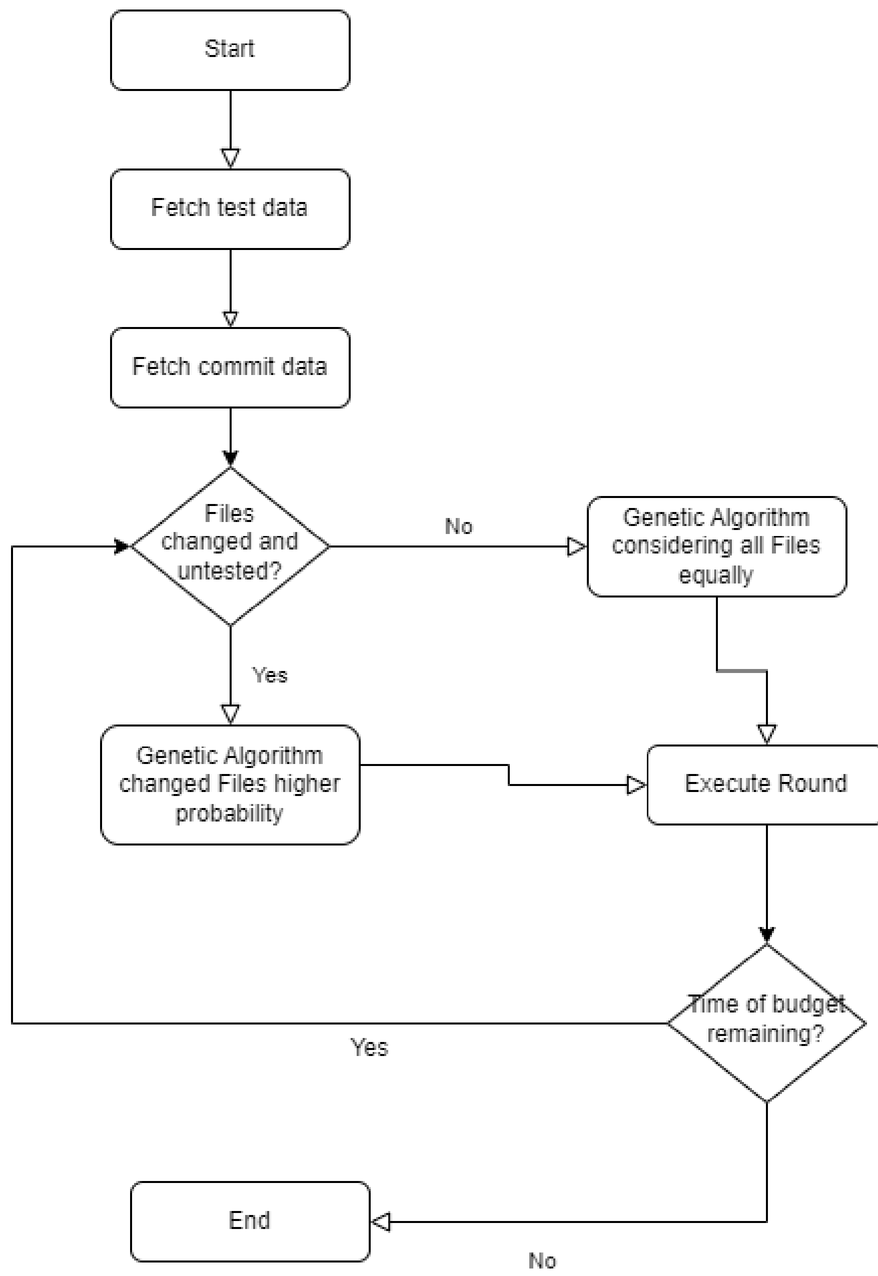


Figure 5.10: Iteration 5: Flow diagram of developed method

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

5.7.3 Treatment Validation And Implementation

It was decided to prototype both approaches to benchmark the approaches. The issue was that none of the ideas fixed the problem of keeping the time budget. Therefore, to address this issue, the decision was to split the execution of the test cases into at most three sub-pipelines, hoping that this would better fill the time budget. The first sub-pipelines are used to check how fast the machine is compared to the other runners, whose average test time is stored in the database. Then, the second tries to fill the pipeline. The idea is that the overhead factor should be more precise for the third pipeline.

Algorithm 5.4 and 5.5 show how this combined implementation works:

1. Calculate the theoretical time budget of the first and the second run.
2. Retrieve the overhead statistics from the database for the given budget. This contains the average overhead factor; for example, if the pipeline with a time budget of 60 seconds takes 120, then this value would be 2, and if it is 30 seconds, it is 0.5. The standard deviation is the second value returned by the database.
3. Adjust the provided budget with the new value. For the first round, divide by the overhead factor plus two times the standard deviation. The idea was that this should ensure the time budget is not exceeded with the first run.
4. Execute the genetic algorithm for the first time budget.
5. Execute the first round of tests.
6. Read how long the tests took and calculate the actual overhead factor
7. Remove the already executed tests from consideration for the next run.
8. Execute the second run with an adjusted time budget if there is still time.
9. Do the same steps as the second execution for the third one if time is still available.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.4: Execution algorithm with three runs part one

Data: time limit

Result: runs executed and combined duration returned

```
1  const firstTimeBudget = timeLimit * 0.3;
2  let secondTimeBudget = timeLimit - firstTimeBudget;
3  const overheadStatistics = await postgre-
     sAdapter.getOverheadFactorAndStdDeviationForTimeBudget(options.duration);
4  const overheadMultiplyFactor = overheadStatistics.overheadfactor + 2 *
     overheadStatistics.stddeviation;
5  const adjustedFirstTimeBudget = firstTimeBudget / overheadMultiplyFactor;
6  await executeRound(adjustedFirstTimeBudget);
7  const firstRunDuration = JSON.parse(readFileSync(‘$folder/runDuration.csv‘,
     ’utf-8’));
8  let secondRunDuration = 0;
9  let thirdRunDuration = 0;
10 let numOfPipelines = 1;
11 for (const test of bestTestSet)
12  {
13     testsMap.delete(test.name);
14  }
15 // calculate remaining time const firstDeviateFactor = firstRunDuration /
     bestPopulation!.duration;
16 console.log(’deviate factor’, firstDeviateFactor);
17 let secondRun = false;
18 secondTimeBudget /= overheadMultiplyFactor + overheadStatistics.stddeviation;
19 secondTimeBudget += Math.max(adjustedFirstTimeBudget -
     bestPopulation!.duration, 0);
20 secondTimeBudget /= firstDeviateFactor;
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.5: Execution algorithm with three runs part two

```
   Data: time limit
   Result: runs executed and combined duration returned
1  if (secondTimeBudget > 500)
2  {
3     await executeRound(secondTimeBudget);
4     secondRunDuration = JSON.parse(readFileSync(‘$folder/runDuration.csv‘,
       'utf-8'));
5     secondRun = true;
6     numOfPipelines++;
7  }
8  else
9  {
10    secondRunDuration = 0;
11 }
12 const firstAndSecondCombinedDuration = firstRunDuration +
    secondRunDuration;
13 const secondDeviateFactor = secondRunDuration / bestPopulation!.duration;
14 const firstAndSecondCombinedDeviateFactor = (firstDeviateFactor +
    secondDeviateFactor) / 2;
15 console.log('first plus second deviate factor',
    firstAndSecondCombinedDeviateFactor);
16 let thirdTimeBudget = timeLimit - firstAndSecondCombinedDuration;
17 thirdTimeBudget /= overheadMultiplyFactor + overheadStatistics.stddeviation;
18 thirdTimeBudget /= firstAndSecondCombinedDeviateFactor;
19 if (thirdTimeBudget > 500)
20 {
21    thirdRun = true;
22    await executeRound(thirdTimeBudget);
23    thirdRunDuration = JSON.parse(readFileSync(‘$folder/runDuration.csv‘,
       'utf-8'));
24    numOfPipelines++;
25 }
26 return firstRunDuration + secondRunDuration + thirdRunDuration;
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.8 6. Iteration

5.8.1 Implementation Evaluation/Problem Analysis

For the analysis of Iteration 5, the same bugs were used as in the previous evaluation. The number of bugs found can be seen in Table 5.6. One improvement is that each run did find at least one bug, but only the pipelines with the time budget of 5 minutes found more than 1 bug. This is because of inefficiencies with the genetic algorithm. On a close inspection, it seemed that tests affected by the change have a very low probability of being selected just because of numbers: the probability of the genome being true at the position of such a test is very low. The ideas of the previous iterations did not bring the improvements required.

	1 min	2 min	3 min	5 min
1.	1	1	1	1
2.	1	1	1	1
3.	1	1	1	2
4.	1	1	1	1
5.	1	1	1	1
6.	1	1	1	1
7.	1	1	1	2
8.	1	1	1	2
9.	1	1	1	1
10.	1	1	1	1

Table 5.6: Evaluation of iteration 5: Number of bugs found with different time budgets and five bugs introduced

The analysis of the execution times shows that the time budget is still not kept completely. The duration of the pipelines and the accompanying boxplot are visible in Table 5.7 and Figure 5.11. It can be observed that the adherence to the time budget is increased. This can be seen well in the boxplot diagram. However, looking at Table 5.7, it is clear that the variation is still huge and that many pipelines take too much time (e.g., time budget of 5 min) or not enough time (1 min).

The data showed that the sub-runs nearly never hit the intended time budget, not even when the adjusted time budget was used. This comes down to run-to-run variation. Tests can sometimes take up to 10 times longer or shorter than what was stored in the database. Execution times during the nightly run showed that the duration of all tests can vary by almost one minute; this is a problem when aiming to hit a time budget of 60 or 120 seconds.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

	1 min	2 min	3 min	5 min
1.	0m 44s	2m 11s	3m 23s	6m 57s
2.	0m 40s	1m 40s	3m 33s	5m 59s
3.	0m 19s	2m 18s	3m 00s	8m 28s
4.	0m 36s	2m 08s	3m 14s	7m 27s
5.	0m 33s	1m 47s	3m 08s	6m 13s
6.	0m 30s	1m 56s	3m 37s	3m 52s
7.	0m 55s	2m 26s	3m 04s	7m 05s
8.	0m 37s	2m 09s	3m 38s	5m 18s
9.	0m 16s	2m 08s	3m 58s	6m 38s
10.	0m 25s	2m 58s	4m 27s	6m 17s

Table 5.7: Evaluation of iteration 5: Execution times of different time budgets and five bugs introduced

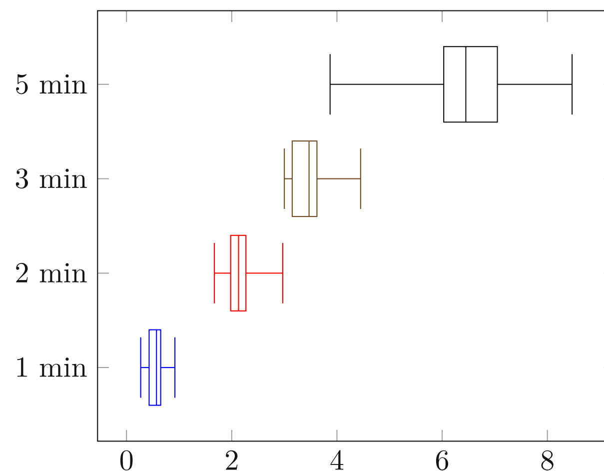


Figure 5.11: Evaluation of iteration 5: Boxplot of execution times from table 5.7

From here, the goal is to get the pipelines to stick closer to the specified time budgets and to find more bugs:

1. It should stick closer to the time budget.
2. It should find more bugs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.8.2 Treatment Design

The design is adjusted: the first decision is not to limit the number of pipelines. This is possible because of the earlier decision to consider only the actual test duration and not count the execution time of the genetic algorithm or the setup time.

The second decision is to consider just tests affected by changes as long as those were not all executed. If there is still time remaining, the goal of the next genetic algorithm is to maximize the coverage of the remaining areas of the code.

The flow of this new approach can be seen in Figure 5.12. The steps associated are:

1. Fetch test and commit data
2. Check if there are files associated with changes and untested.
3. If yes, execute the genetic algorithm with only files associated with changes.
4. If not, execute the genetic algorithm considering all files which have to be only files not associated with changes.
5. Execute the selected test cases.
6. Check if there is still time remaining.
7. If yes, return to step 2.
8. If no, finish execution.

Another consideration is to check whether genetic algorithm implementation performed poorly. Therefore, different frameworks that allow genetic algorithm implementation are investigated. The geneticalgorithm[7] package was selected. The proposed design changes are presented in Figure 5.13.

[7] https://www.npmjs.com/package/geneticalgorithm, accessed on: 01.05.2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

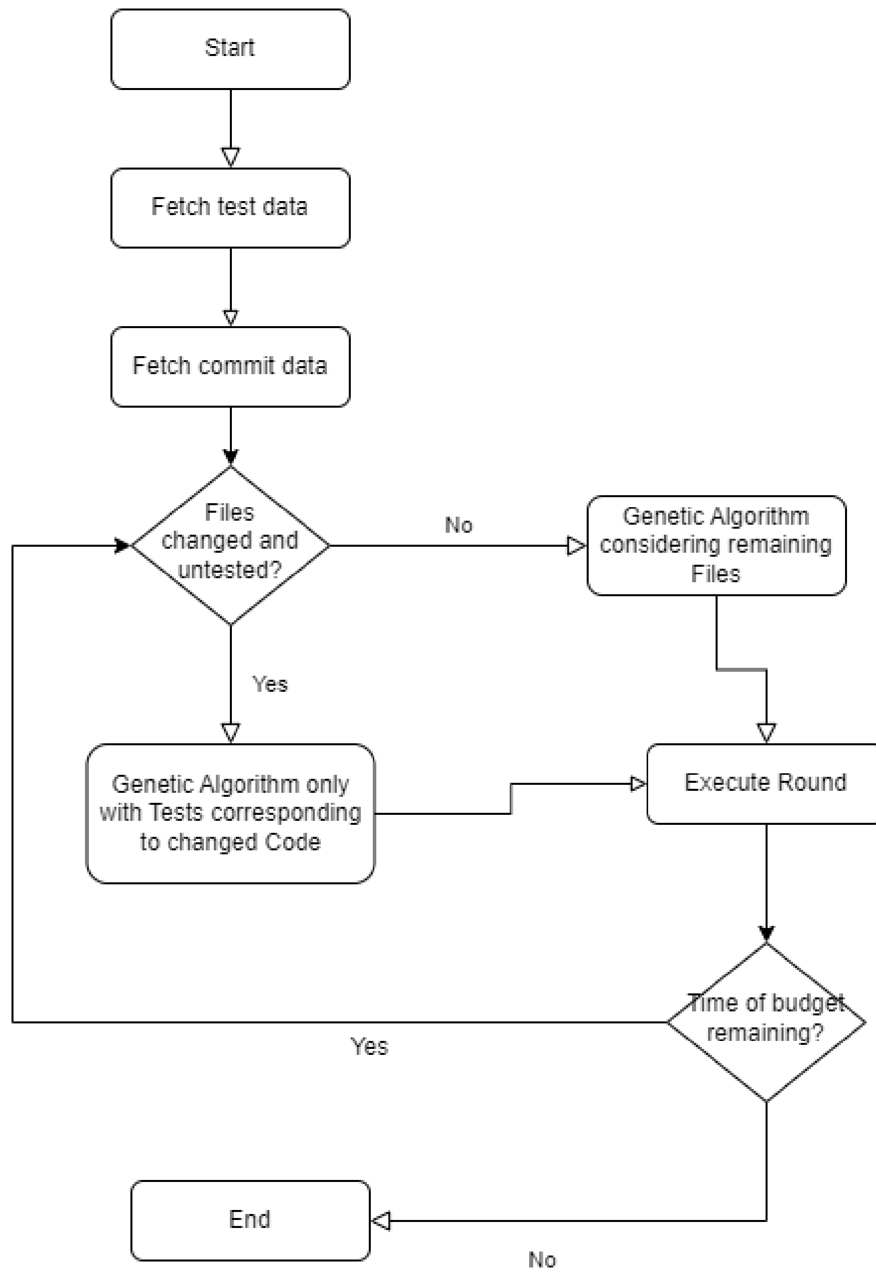


Figure 5.12: Iteration 5: Flow diagram of developed method

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

This design mainly encompassed the following changes:

1. Store when the last fitness changes was. This is now used to determine when to terminate.
2. Move fitness function out of a separate file and into a genetic algorithm function.
3. provides a "doesABeatBFunction" which was needed for the framework.
4. Implement all necessary changes for the introduction of the framework.

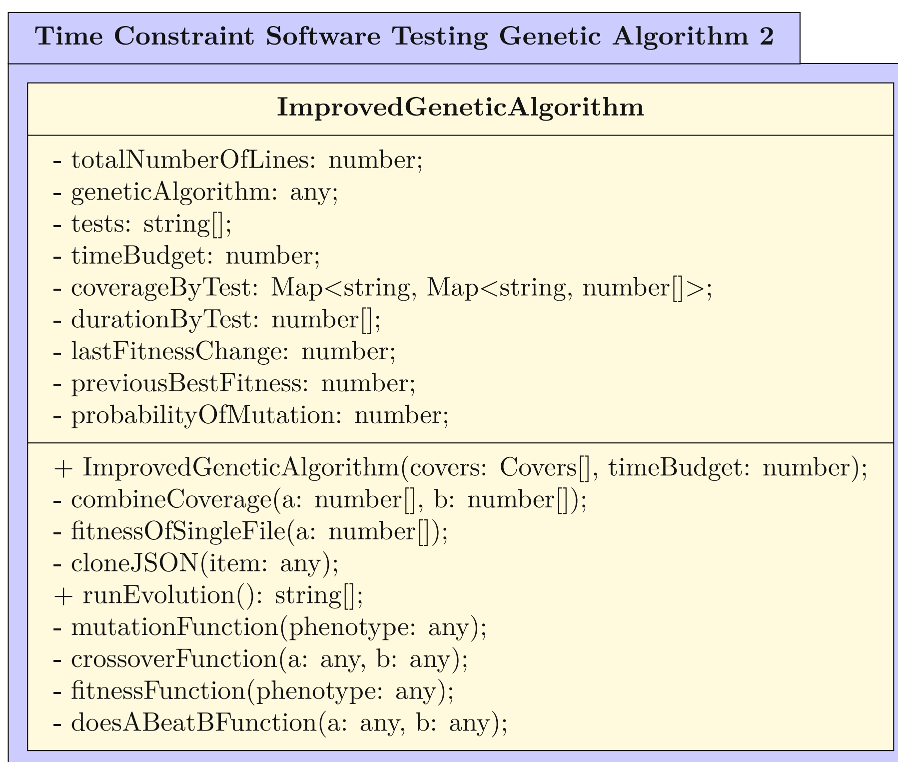


Figure 5.13: Iteration 5: PostgresAdaper design

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.8.3 Treatment Evaluation

During the evaluation of the last iteration, one interesting aspect was the overshooting of the pipelines, especially with a time budget of 5 minutes. This issue was investigated during the design phase.

This behavior appears if a high number of tests are selected where the duration during the nightly run is lower than the observed time. This is because the conditions with the runners were ideal during the night; however, because not each runner is a dedicated machine, if many pipelines were run in parallel the execution time can vary drastically. This was mentioned before, with tests taking up to ten times more or less during the nightly run.

There were two potential solutions to address this issue. The first option is to write a custom test runner to stop test execution when the time budget is filled. This means making the time budget strict. This would be the ideal fix, but it was decided this issue is out of scope because of the time it would take to implement. The second option is to split the pipelines into more sub-pipelines. However, the overhead of selecting tests would take too much time at this point. Exceeding the time used for the time budget. If this method were to be used as a general framework, the first solution would need to be implemented. The decision is to accept the current behavior and just focus on the test selection.

5.8.4 Treatment Implementation

A few steps are key for adapting to the new framework. The first one is the new fitness, presented in Algorithm 5.6. The main differentiator to the previous iteration is the simplicity of the implementation. Because only either one or all of the files the algorithm receives are changed, no tuning is required. The fitness is now -1 if the time budget is exceeded (see line 18). Other than that, it just sums up the coverage of each file. The calculation of this can be seen in the fitnessOfSingleFile function declared in line 28.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.6: Fitness functions of new genetic algorithm

```
1  private fitnessFunction(phenotype: any)
2  {
3      let sumOfTime = 0;
4      let coverageByFile = new Map<string, number[]>();
5      for (let i = 0; i < phenotype.genome.length; i++)
6      {
7          if (phenotype.genome[i])  const testname = this.tests[i];
8          sumOfTime += this.durationByTest[i];
9          const coveredFiles = this.coverageByTest.get(testname) ?? new Map();
10         for (const filename of coveredFiles.keys())
11         {
12             const previousFileCoverage = coverageByFile.get(filename) ?? [];
13             coverageByFile.set(filename,
                 this.combineCoverage(previousFileCoverage,
                 coveredFiles.get(filename)));
14         }
15     }
16     if (sumOfTime > this.timeBudget)
17     {
18         return -1;
19     }
20     if (coverageByFile.size == 0)
21     {
22         return 0;
23     }
24     return Array.from(coverageByFile.values())
25     .map((x) => this.fitnessOfSingleFile(x))
26     .reduce((x, y) => x + y);
27 }
28 private fitnessOfSingleFile(a: number[])
29 {
30     return a.reduce((a, b) => a + Math.min(b, 1)) / a.length;
31 }
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

The crossover function is also crucial to the implementation presented in Algorithm 5.7. All that is done here is swapping the arrays' first half between the genomes. This implementation is simple and might be sufficient but could be changed in the next iteration if needed.

Algorithm 5.7: Crossover function of new genetic algorithm

```
1  private crossoverFunction(a: any, b: any)
2  {
3      let x = this.cloneJSON(a);
4      let y = this.cloneJSON(b);
5      for (let i = 0; i < this.tests.length / 2; i++)
6      {
7          const swap = x[i];
8          x[i] = y[i];
9          y[i] = swap;
10     }
11     return [x, y];
12 }
```

Algorithm 5.8 shows how the coverage of two different genomes is combined.

Algorithm 5.8: Combine coverage helper function used by crossover

```
1  private combineCoverage(a: number[], b: number[])
2  {
3      const result = []; for (let i = 0; i < a.length || i < b.length; i++)
4      {
5          result.push((a[i] ?? 0) + (b[i] ?? 0));
6      }
7      return result;
8  }
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.9 shows the improved genetic algorithm's new finishing function. Notable here is that the program keeps track of how the fitness value changes. If the fitness value does not change for 50 iterations, the execution is terminated. The sophisticated implementation allowing each iteration to be faster this means that over 1000 iterations required just seconds in the trial experiments.

Algorithm 5.9: Finish function of new genetic algorithm

```
1  private finished()
2  const bestPhenotype = this.ga.best();
3  const newBestFitness = this.fitnessFunction(bestPhenotype);
4  if (newBestFitness == 0)
5   {
6      return false;
7   }
8  if (newBestFitness === this.oldBestFitness)
9   {
10     this.lastFitnessChange++;
11     if (this.lastFitnessChange > 50)
12      {
13         return true;
14      }
15  }
16 else
17  {
18     this.oldBestFitness = newBestFitness;
19     this.lastFitnessChange = 0;
20  }
21 return false;
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.9 7. Iteration

5.9.1 Implementation Evaluation/Problem Analysis

A pipeline with the 5 bugs and the improved algorithm was triggered. The durations (Table 5.8) show that the time budget is well-filled. The overshoot mentioned in the previous treatment evaluation can be observed. The biggest overshoot was 24 seconds (2 min budget 9. run). How close the algorithm now sticks to the time budget can be observed in the boxplot. It is the widest for two minutes, but even there, it is not as wide as in the previous iterations.

	1 min	2 min	3 min	5 min
1.	0m 59s	1m 59s	3m 12s	4m 59s
2.	1m 01s	2m 17s	3m 11s	4m 59s
3.	0m 59s	2m 20s	2m 59s	4m 59s
4.	0m 59s	2m 10s	3m 10s	5m 13s
5.	1m 05s	2m 08s	2m 59s	5m 00s
6.	0m 59s	2m 12s	3m 11s	5m 15s
7.	1m 00s	2m 13s	3m 12s	5m 08s
8.	1m 02s	1m 59s	3m 12s	5m 12s
9.	1m 08s	2m 24s	2m 59s	5m 00s
10.	1m 05s	2m 10s	2m 59s	4m 59s

Table 5.8: Evaluation of iteration 6: Execution times of different time budgets and five bugs introduced

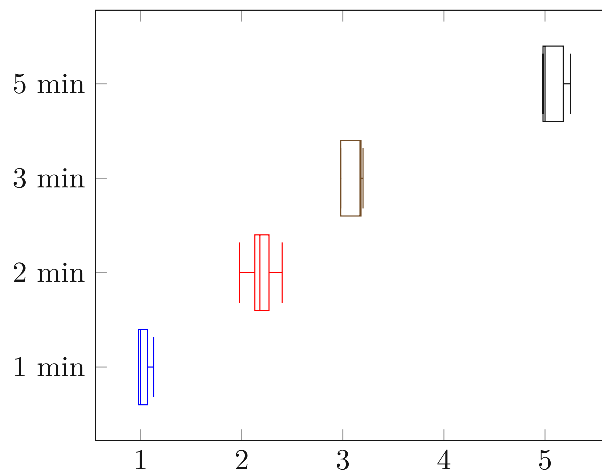


Figure 5.14: Evaluation of iteration 6: Boxplot of execution times from table 5.8

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 5.9 shows the number of bugs found. Compared with previous iterations, this is a complete success. In each run, all bugs were found.

	1 min	2 min	3 min	5 min
1.	5	5	5	5
2.	5	5	5	5
3.	5	5	5	5
4.	5	5	5	5
5.	5	5	5	5
6.	5	5	5	5
7.	5	5	5	5
8.	5	5	5	5
9.	5	5	5	5
10.	5	5	5	5

Table 5.9: Evaluation of iteration 6: Number of bugs found with different time budgets and five bugs introduced

Following the implementation evaluation, it was decided that the implementation is sufficient for demonstrating practicability and drawing conclusions. For this, it is still necessary to implement an algorithm for Random Testing as a benchmark to verify the feasibility of the improved genetic algorithm presented in Iteration 6. Therefore, the requirements for this iteration are:

1. It should supply a random algorithm that can be used as a control against the improved genetic algorithm.

5.9.2 Treatment Design

The idea for a RT is to select a test using a random index and then sum up the supposed time duration for all tests selected. No additional tests are be selected if the supposed time budget is exceeded. The design for this can be seen in Figure 5.15.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

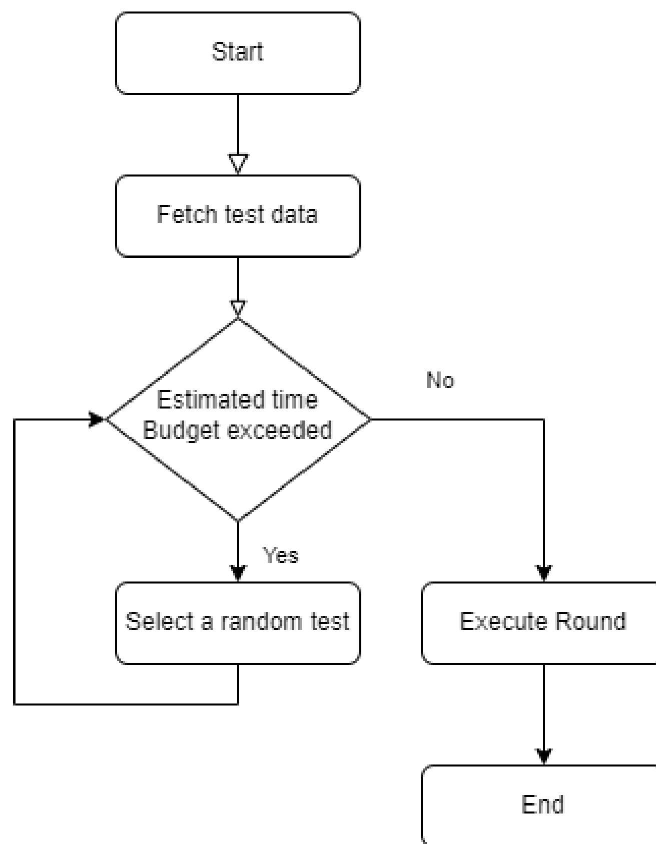


Figure 5.15: Iteration 7: Flow Diagram of RT approach

5.9.3 Treatment Evaluation and Treatment Implementation

Due to the simplicity of the approach it was decided to implement it completely and see how well it adheres to the time budget, then evaluate it in the next iteration and improve it if needed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.10 8. Iteration

5.10.1 Implementation Evaluation/ Problem Investigation

Because of the randomness of the RT algorithm, introducing bugs made no sense for the evaluation. Therefore, the pipelines were triggered with different time budgets and observed how well the implementation stuck to it.

	1 min	2 min	3 min	5 min
1.	3m 11s	3m 45s	3m 28s	6m 42s
2.	3m 40s	4m 27s	6m 10s	6m 37s
3.	2m 47s	4m 29s	6m 11s	5m 40s
4.	3m 17s	4m 03s	5m 37s	6m 44s
5.	4m 02s	3m 21s	7m 04s	6m 33s
6.	2m 57s	3m 49s	6m 14s	5m 28s
7.	3m 11s	4m 22s	6m 02s	6m 39s
8.	3m 10s	3m 43s	5m 39s	6m 55s
9.	2m 55s	3m 44s	6m 11s	6m 30s
10.	3m 18s	4m 35s	6m 24s	5m 06s

Table 5.10: Evaluation of iteration 7: RT approach: Evaluation of keeping time budget

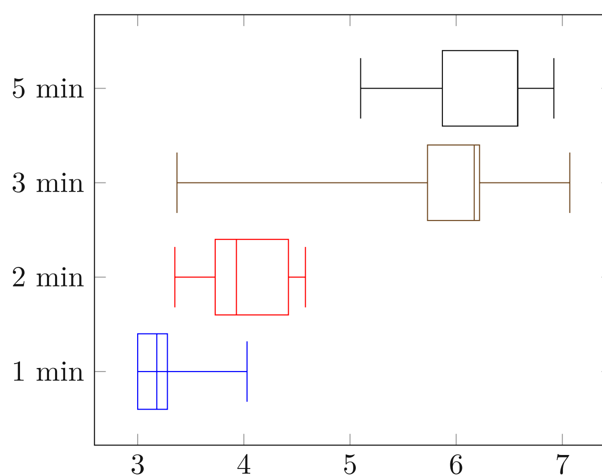


Figure 5.16: Evaluation of iteration 8: RT approach: Boxplot diagram of execution times from table 5.10

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Table 5.10 shows the execution times, and Figure 5.16 is a box plot of the data from the table. It is clear that this implementation is not sufficient. If comparing an execution taking about 1 minute from the improved algorithm and 3 minutes from the random testing algorithm it is not a fair comparison.

The main objective for this stage is to make the RT stick to the time budget. This results in the new requirement:

1. The Random Testing (RT) should adhere to the time budget.

5.10.2 Treatment Design

Several changes were made to the design of the Random Testing algorithm to fulfill the requirements. The new flow of the algorithm is presented in Figure 5.17. It includes the following changes:

1. Overhead Statistics: The improved version now uses the overhead statistics to better address the run-to-run variation of test execution times.
2. Multiple runs: If the time is not used up, another run with the remaining time is started.
3. Splitting of Run: Just like it was decided for the genetic algorithm, the pipeline shall always be split into multiple sub-runs. It was decided only to use a fourth of the available time budget.

5.10.3 Treatment Evaluation and Implementation

This approach will presumably work because it has already been used for the improved genetic algorithm. Therefore, there is nothing to evaluate, and the changes were implemented.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

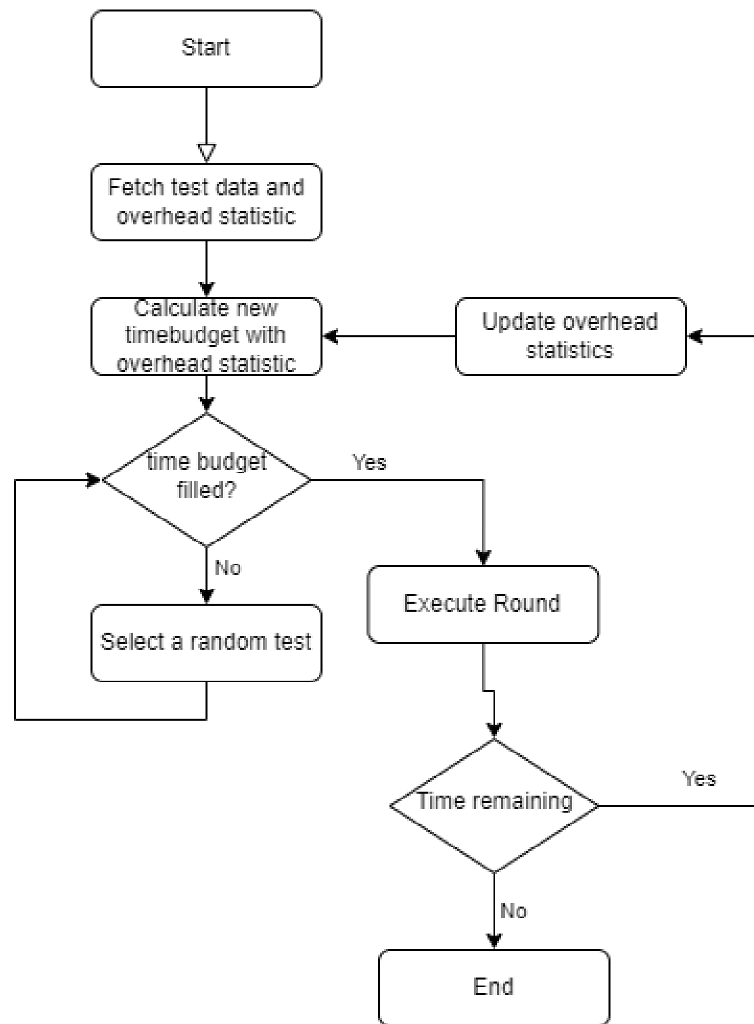


Figure 5.17: Iteration 8: RT approach: Improved flow diagram

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

5.11 9. Iteration

5.11.1 Implementation Evaluation

Another pipeline was triggered with the improved Random Testing approach. The results are presented in Table 5.11 and Figure 5.18. While there is still some variation between runs, this ranges in the same time frames as the improved genetic algorithm. Based on this similarity of variations, the decision is that the implementation is finished and to continue with the overall evaluation.

	1 min	2 min	3 min	5 min
1.	0m 56s	1m 52s	2m 45s	4m 36s
2.	1m 12s	1m 51s	2m 42s	4m 30s
3.	0m 56s	1m 51s	2m 46s	4m 52s
4.	1m 01s	2m 02s	3m 24s	4m 38s
5.	0m 56s	1m 57s	2m 44s	4m 34s
6.	0m 54s	1m 50s	2m 50s	4m 30s
7.	0m 58s	1m 48s	3m 06s	5m 00s
8.	0m 54s	1m 49s	2m 43s	4m 55s
9.	1m 10s	1m 48s	2m 42s	4m 30s
10.	0m 59s	1m 50s	2m 43s	4m 55s

Table 5.11: Evaluation of iteration 8: RT Approach: Evaluation of keeping time budget

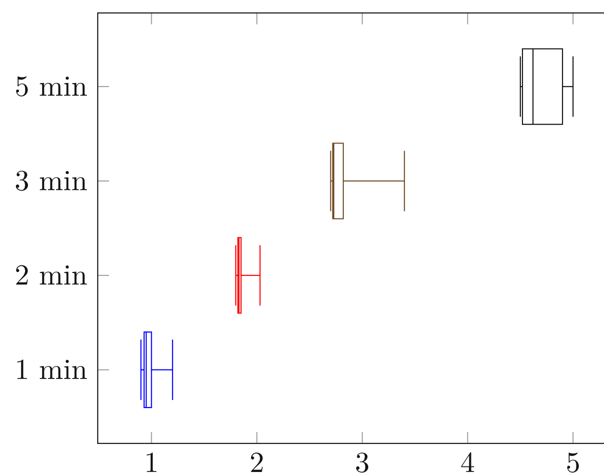


Figure 5.18: Evaluation of iteration 8: Boxplot of execution times from table 5.11

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Evaluation and Results

This chapter is split into multiple parts. The first section compares the algorithm with RT as a benchmark with different commit sizes. The second part focuses on evaluating different numbers of bugs with varying sizes of commits. For both of these sections, a specified number of bugs is introduced, and in addition, files are changed in a way that they are considered as changed but have no bugs introduced. For this, empty lines are utilized to increase the number of files changed and include these files to be covered. To conclude, there is a discussion about the optimal time budget.

6.1 Benchmark Validation

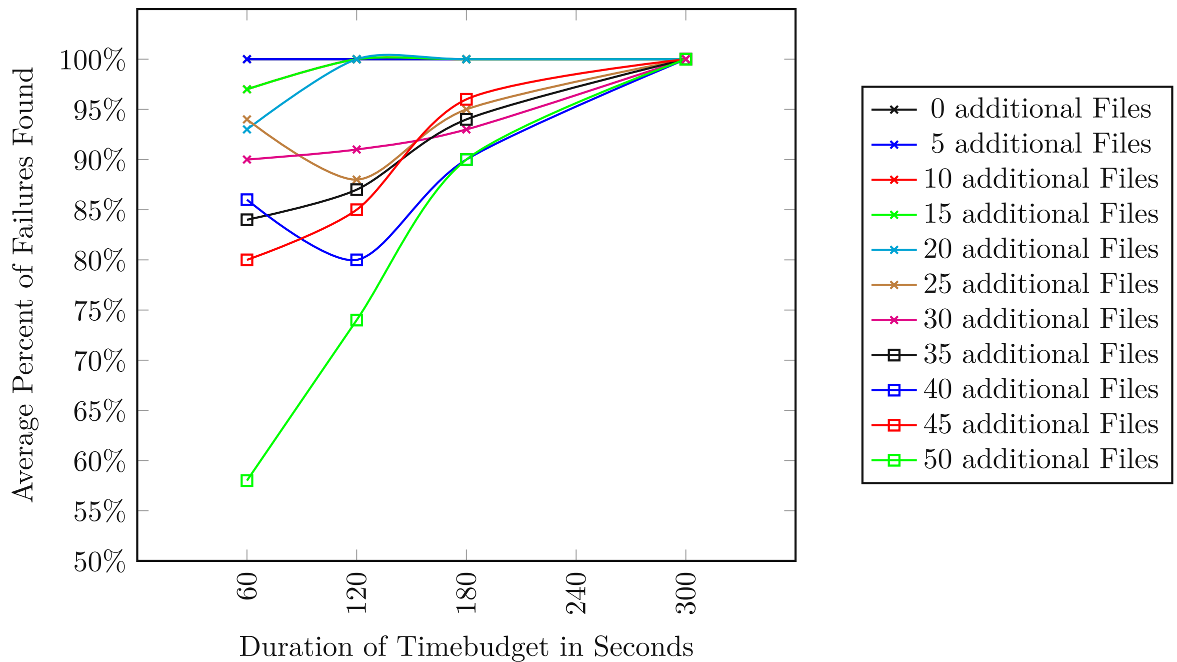
This section is focused on analyzing the algorithm by comparing it to Random Testing. A pipeline with one bug and different commit sizes was used for this. The file labels in the graphs (Figures 6.1 and 6.2) indicate how many files were changed besides the bug. For the changed files, empty lines were added to be considered changed, but the changes do not trigger new bugs. The algorithm does not differ between files where functionality was changed and files where just an empty line was added. This means that the total commit size for ten files and one bug is eleven. This labeling is relevant for all graphs and tables for this chapter.

Each type of pipeline was run 100 times. This means that Tables 6.1 and 6.2 show directly the number of pipelines that found bugs. This number was taken from the CI system.

Table 6.1: Developed method: Percentage of pipelines finding the introduced bug with x additional files in commit

	1 min	2 min	3 min	5 min
0 additional Files	100%	100%	100%	100%
5 additional Files	100%	100%	100%	100%
10 additional Files	97%	100%	100%	100%
15 additional Files	97%	100%	100%	100%
20 additional Files	93%	100%	100%	100%
25 additional Files	94%	88%	95%	100%
30 additional Files	90%	91%	93%	100%
35 additional Files	84%	87%	94%	100%
40 additional Files	86%	80%	90%	100%
45 additional Files	80%	85%	96%	100%
50 additional Files	58%	74%	90%	100%

Figure 6.1: Developed method: Percentage of pipelines finding the introduced bug with x additional files in commit



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 6.1 shows the numbers for the improved version of the genetic algorithm. Noticeable is that with a commit size of 30, the bug was still found in 90 percent of pipelines. It can be seen that with a commit size of 50, the results dropped quite drastically compared to 45, however they are still above 50 percent for all pipelines.

Figure 6.1 presents the results as a graph. The graph illustrates how the approach performs, except for the 50 additional files line, it never drops below 80 percent.

The random testing starkly contrasts the approach created in the method design chapter (chapter 5). The graph (Figure 6.2) almost resembles a line. Statistically, this is expected. For each duration, the expected number of pipelines finding the introduced bug is calculated by:

$$E(T_b) = \frac{T_b * 100}{T_t}$$

Where $E(T_b)$ is the expected average number of pipelines finding the bug for a given time budget, T_b is the time budget, and T_t is the total execution time for the whole test suite. As an example, for a total pipeline of about 8 minutes (for the case study project, it is seven to nine minutes for all unit tests), it's about 12.5 percent for a 1-minute budget.

$$E(1) = \frac{1 * 100}{8} = 12.5$$

For a 5-minute time budget, it is 62.5 percent.

$$E(5) = \frac{5 * 100}{8} = 62.5$$

These are the numbers represented in the graph. The detailed numbers shown in Table 6.2 seem to support this.

The developed approach works well based on the available data, given that there is enough time to fit all tests associated with changes into the time budget. However, the numbers seem to indicate that the coverage is still reasonable even if not all tests can be fitted into the time budget since the genetic algorithm seems to still provide good coverage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 6.2: Random testing: Percentage of pipelines finding the introduced bug with x additional files in commit

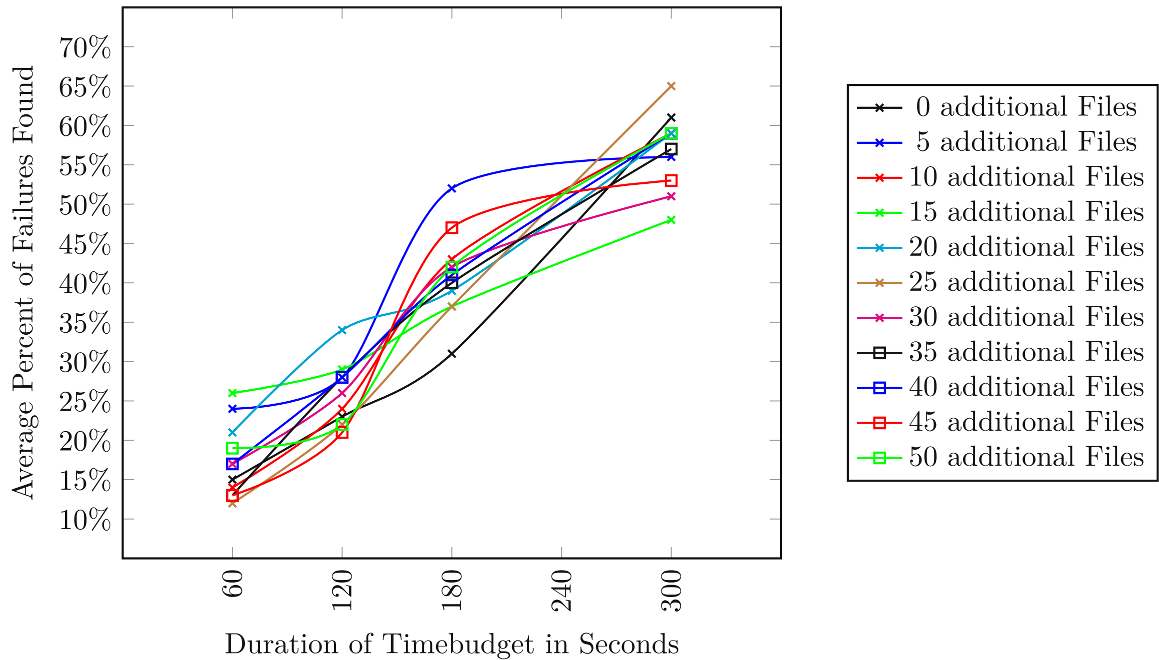


Table 6.2: RT approach: Percentage of pipelines finding the introduced bug with x additional files in commit

	1 min	2 min	3 min	5 min
0 additional Files	15%	23%	31%	61%
5 additional Files	24%	28%	52%	56%
10 additional Files	14%	24%	43%	59%
15 additional Files	26%	29%	37%	48%
20 additional Files	21%	34%	39%	59%
25 additional Files	12%	22%	37%	65%
30 additional Files	17%	26%	42%	51%
35 additional Files	13%	28%	40%	57%
40 additional Files	17%	28%	41%	59%
45 additional Files	13%	21%	47%	53%
50 additional Files	19%	22%	42%	59%

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.2 Commit Size Analysis

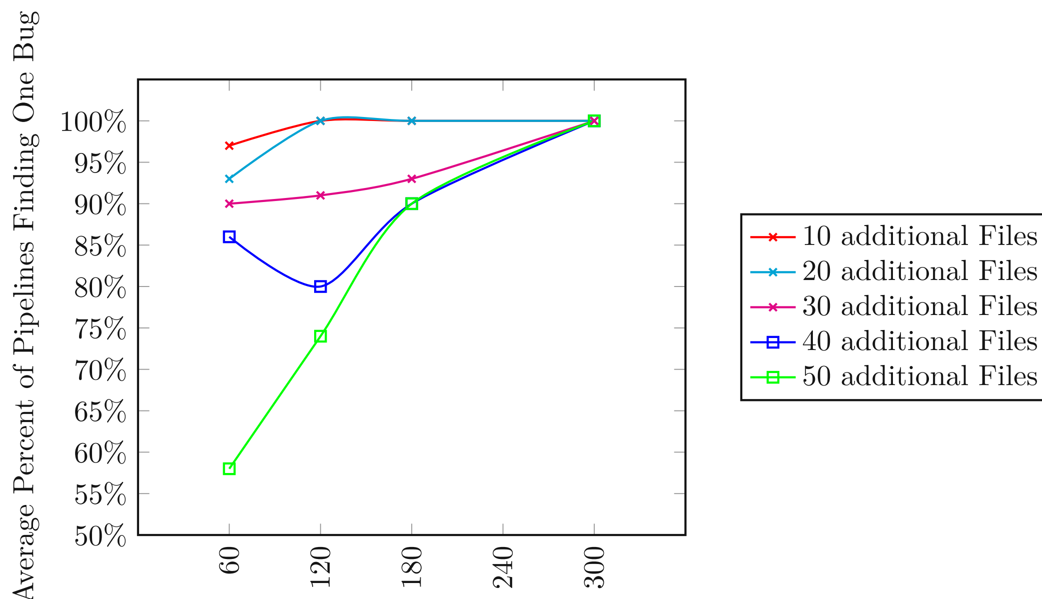
This section tests the algorithm's performance when different numbers of bugs are introduced with varying sizes of commits.

Figure 6.3 shows the algorithm's performance when one bug is introduced. This graph is the same as 6.1 and serves as a baseline. It is observable from Figures 6.4 to 6.6 that the number of pipelines that find a bug does increase. The only exception where the effectiveness did not changed is the three bugs graphs. It seems that this is because the code section which was changed is not covered properly.

Figure 6.6 shows that the pipeline's effectiveness is over 80 percent, even with a commit size of 50 for five introduced bugs. This consistent increase indicates that the algorithm's effectiveness increases with additional files. The conclusion is that the worst case is the baseline of a single bug introduced into the codebase.

The graphs in Figures 6.7 to 6.10 indicate that if multiple bugs are introduced, the approach can find all of them. The only bug not detected is the third one, which was not covered by the code coverage. The bad news is that not all bugs are found within the 60-second time budget. Most of the five introduced bugs are only found within a time budget of three minutes.

Figure 6.3: Improved algorithm: Percentage of pipelines finding at least one bug with one bug introduced plus x additional files in commit



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 6.4: Improved algorithm: Percentage of pipelines finding at least one bug with two bugs introduced plus x additional files in commit

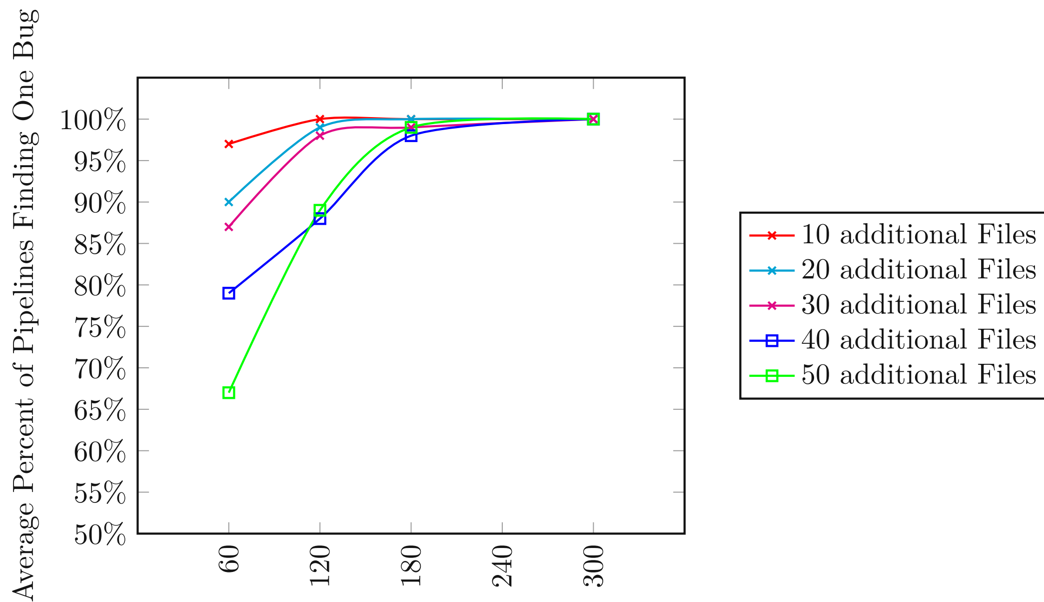
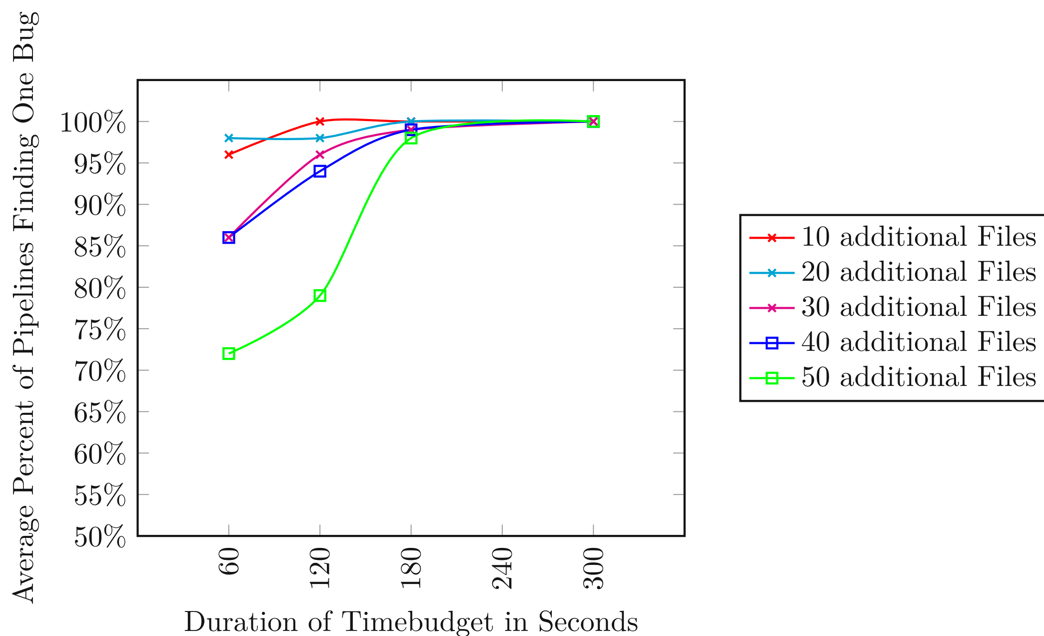


Figure 6.5: Developed method: Percentage of pipelines finding at least one bug with three bugs introduced plus x additional files in commit



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 6.6: Developed method: Percentage of pipelines finding at least one bug with five bugs introduced plus x additional files in commit

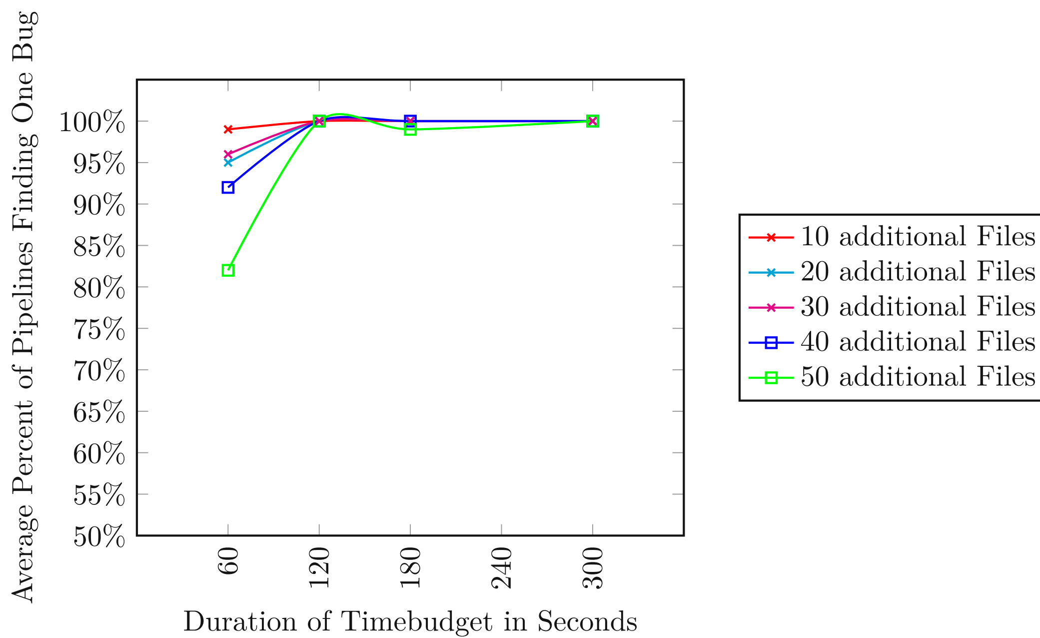
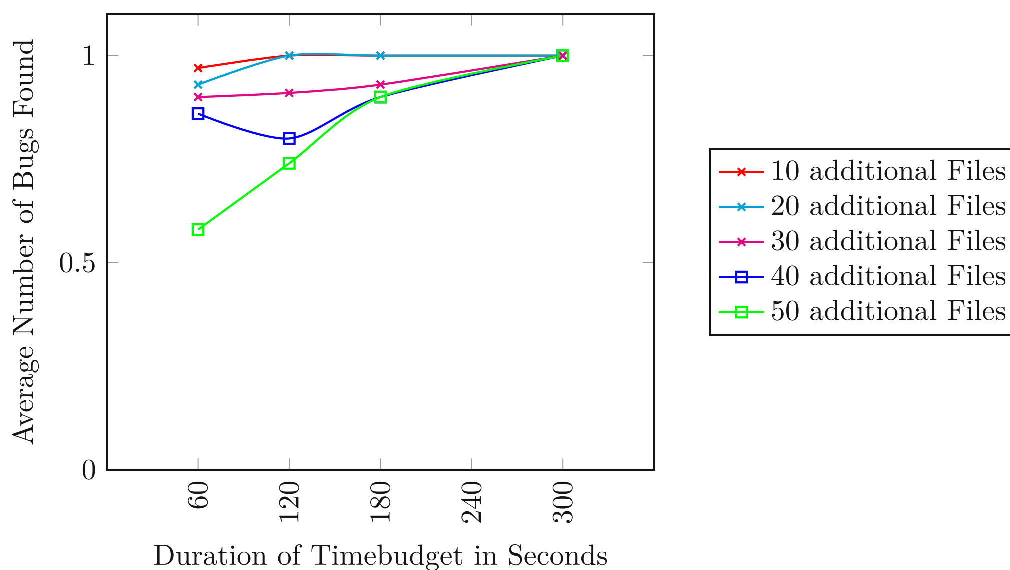


Figure 6.7: Developed method: Number of bugs found on average with one bug introduced plus x additional files in commit



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 6.8: Developed method: Number of bugs found on average with two bugs introduced plus x additional files in commit

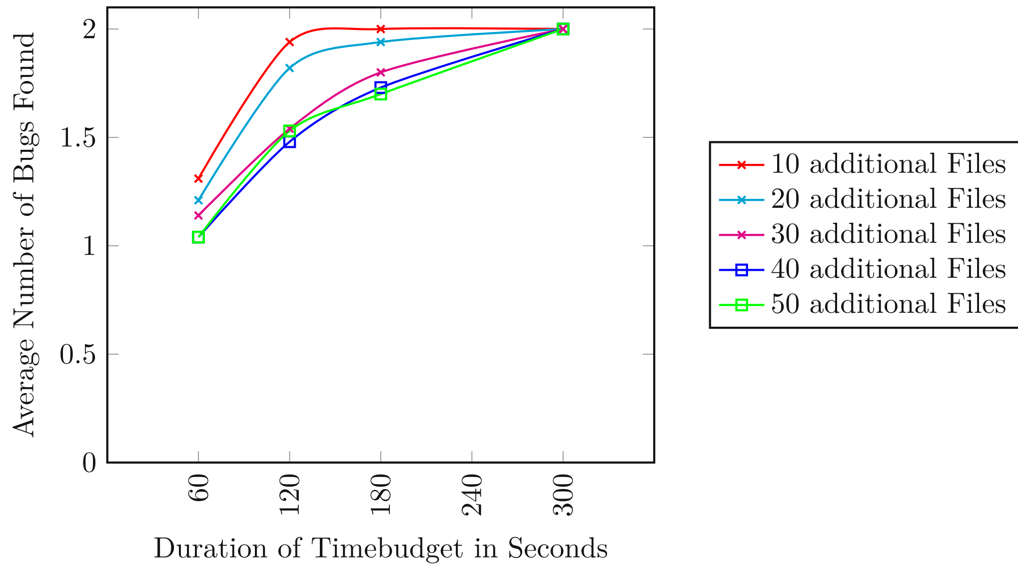
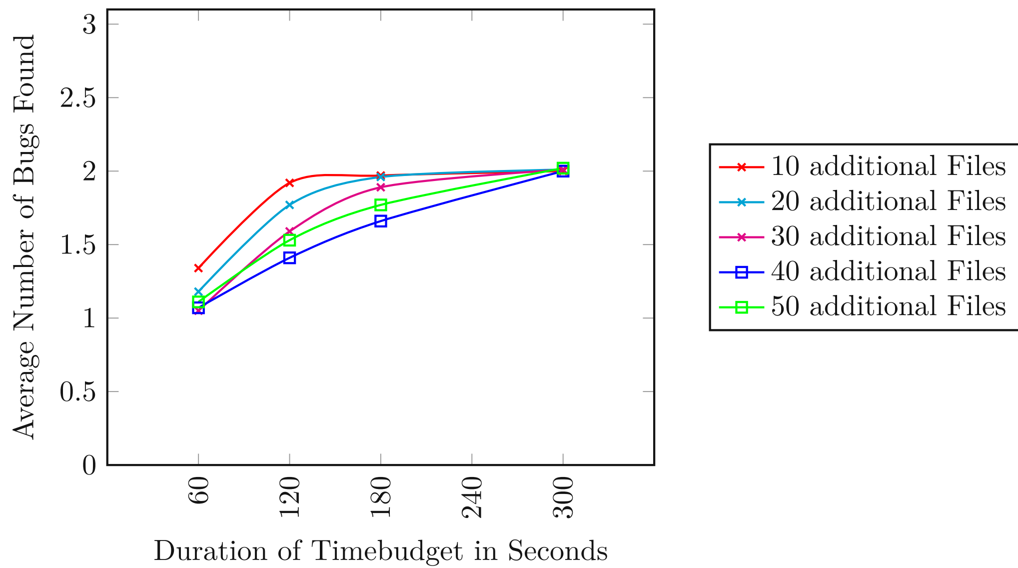
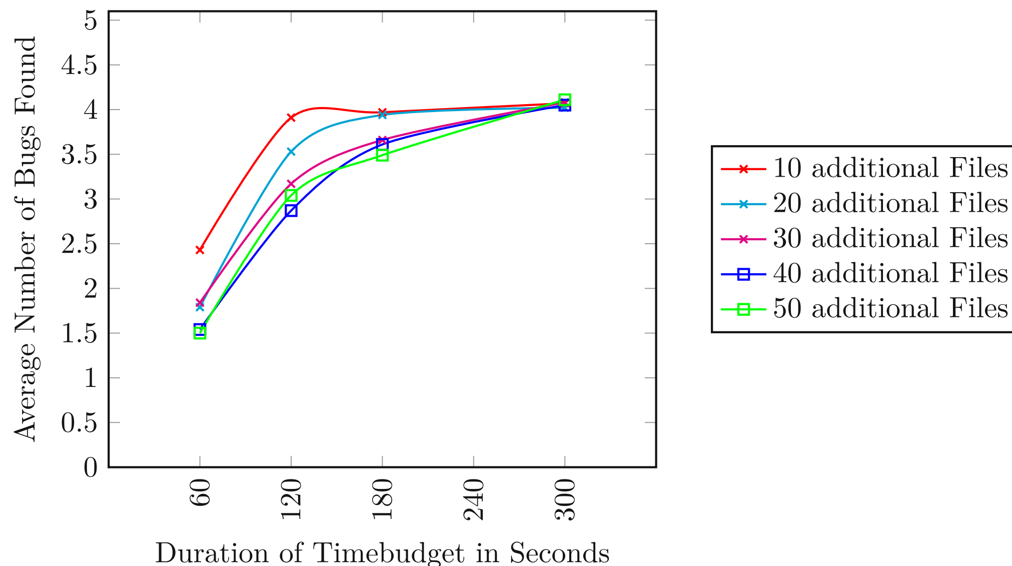


Figure 6.9: Developed method: Number of bugs found on average with three bugs introduced plus x additional files in commit



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 6.10: Developed method: Number of bugs found on average with five bugs introduced plus x additional files in commit



6.3 Ideal Time Budget

The data indicates that an approach limiting execution time to a time budget that there is no ideal universal time budget. From the implementation and also the evaluation, some factors can be presented to have an impact on how high such a time budget should be:

1. Expected Bug-to-Noise Ratio: Depending on the number of bugs expected to be in the code and the number of comparative noise, the time budget needs to be adapted. Furthermore, as shown in the Bug Evaluation section, it might be that only a few or all bugs are found depending on the time budget. This means it must also be considered if all bugs should be found or if it is sufficient to find at least one.

2. Number of additional Files: Of course, the number of files in a project has an impact. This comes down to the likelihood of detecting a bug. The commit size analysis section might help when looking at this factor.

3. Coverage of Remaining Code: One factor considered only as a side note is also important to the coverage over the whole project. The developed approach tries to maximize the coverage over the whole project if time is still available. Due to time constraints, no investigations were made on how well the approach performs in that regard; it is expected that if good coverage of the remaining code needs to be achieved, the time budget needs to be higher.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Discussion

This chapter discusses the findings concerning the research question.

7.1 Research Question 1

The first research question is: "What characteristics and criteria should be considered in developing a test case selection algorithm that effectively determines the tests within a suite that are susceptible to the impact of pushed commit(s)?"

This question is focused on the metrics available for the development of the time-constraint testing approach. The literature research, as well as the investigation into jest, presented different choices. In this regard, the following findings were achieved:

1. Dependencies via File Import: The first characteristic explored is dependencies. Dependency relations were retrieved from the file imports of source code files. Each dependency was read from the file consuming the code. With this, an association between a provider and a consumer is possible. A provided file offers some code, like a function, and a consumer imports and executes the function. The tested approach was to backtrack from a file changed in a SCM commit to the test cases. This is done by analyzing which files this changed file consumed functionality from. Then, make the same association for the producer file. Repeat this process until either a test file is reached or a circle is detected.

   The findings for this approach utilizing dependencies are that test files can be found this way, but the import characteristic alone is too unspecific. It returned too many test files, some of which did not touch the code area.

2. Total Coverage: This metric, described in several papers [58], aims to maximize the coverage over the whole code of a project. This approach was tested with the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

genetic algorithm and included a higher fitness value for test cases associated with changes.

Analysis during the development of the time-constraint method showed that this characteristic is not precise enough to reliably find errors introduced into the codebase.

3. Per File Changed Coverage Association: The characteristic employed in the final version of the method is the association of changed files with test files that cover each changed file.

   Testing for the iterations showed that when this association was leveraged, the best results were achieved among all tested approaches. For this, the coverage of the changed files was maximized, and if possible, all test files associated with changes were eventually selected.

7.2 Research Question 2

The second research question: "Which of the determined tests are feasible and essential to execute within a given time constraint in order to maximize bug detection?" is investigated in the method design chapter (chapter 5). An approach that answers this question was developed. The developed approach selects the tests employing the workflow presented in iteration six. It consists of three parts:

1. Pre-Loop Phase: This phase (shown in Figure 7.1) fetches the data required for selecting test cases. This includes:
   a) Test Data: The run information stored inside the database of execution times during the last nightly run and the coverage on a per-test file basis.
   b) Overhead Statistics: Consists of two metrics: The average duration and the standard deviation of the time constraint pipeline executions with the same time budget.
   c) Commit Data: The commit information retrieved from the SCM system since the branch was forked from the main branch.
2. Method Loop: This section repeatably selects and executes the test cases until the time budget is filled. The steps presented in Figure 7.2 are:
   a) Adjust Time Budget: In the first step, the overhead statistics are used to adjust the time budget. For this, a deviation factor is calculated. This factor is the difference between the last observed run during the nightly and the average plus two times the standard deviation of the time constraint runs. Then, the time budget is adjusted by this factor. Furthermore, it is divided into three parts, which increases the number of sub-runs and shows better adherence to the time budget.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

b) Test Case Selection and Execution: This is described in its dedicated part. It selects and executes the tests.

c) Continue Run Decision: Following test execution, a decision is made to continue the loop if time is left or otherwise terminate.

d) Update Overhead Statistics: A new overhead factor is calculated to adjust the remaining time. The supposed time according to the nightly pipeline is divided by the observed time for test execution.

3. Test Case Selection and Execution: In this section, shown in Figure 7.3, the genetic algorithm for the test case selection and the selected test suite is executed. It includes the sub-steps:

   a) Changes Left Check: At first, there is a check whether all the test files associated with changes, this means that they cover a changed file, have already been executed. If so, the genetic algorithm is run with all tests. Otherwise, only with tests that have coverage over the changed files.

   b) Genetic Algorithm with additional Files Associated with Changes or All additional Files: The implementation of the genetic algorithm is the same regardless of the test set provided to it. The algorithm encodes the available test files as a genome, and the fitness is calculated as the selected coverage over the possible coverage in a binary fashion. This means it is not considered how often a line is selected; it is just that it is. This was done as a simplification. The initial populations are created randomly. The genetic algorithm terminates when the highest observed fitness does not change for 50 evolutions. However, at least 1000 evolutions are run.

   c) Execute Round: Executes all test files the Genetic algorithm selects.

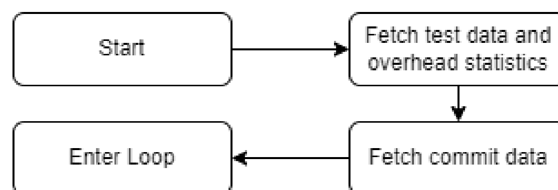


Figure 7.1: Pre-loop phase of the developed method

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

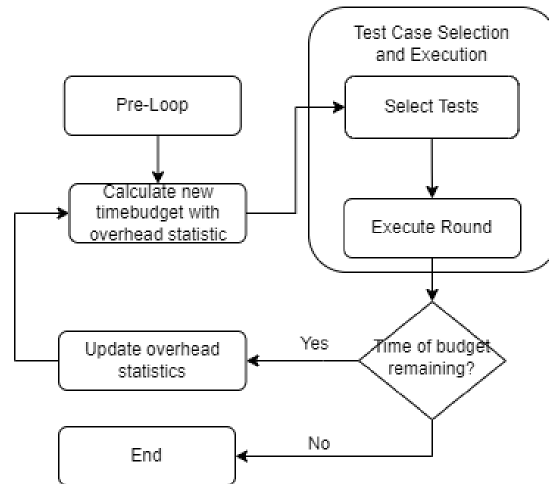


Figure 7.2: Loop phase of the developed method

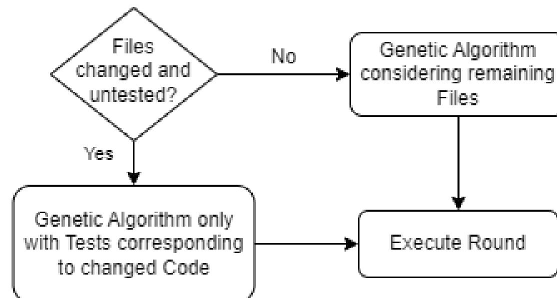


Figure 7.3: Test selection and execution of the developed method

7.3 Research Question 3

The answer to the third research question "How many bugs in the code does such an approach find compared to running the whole test suite?" was presented in the method design (chapter 5) and the evaluation chapter(chapter 6). This research question can only be addressed with experimental evidence. However, the results presented in Table 6.9 indicate that it is possible for the time-constraint approach to reveal all bugs that were introduced into the code. Compared to the evaluation done during method development, the bugs introduced for the bug evaluation section of the evaluation chapter indicate that while it is possible to find all bugs, this is not always the case. For the 8 bugs introduced pipeline, it can be seen that the more other files are changed, the fewer bugs can be found.

For the time-constraint approach, the following reasons lead to a lower bug detection rate.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

1. Coverage: In order to find a bug, it must be covered. This means that if a bug is not found with the full test suite, it can also not be found with the shortened one.
2. Time-Budget: As was shown in the Bug Evaluation, if not all files that cover a test fit within the time budget, then not all of them get executed. This results in a reduction in coverage, and a bug might not be found.
3. Commit-Size: The second variable determining how many tests need to be covered within the time budget is the number of files that have been changed. So if the number of files changed is too high, then it cannot be guaranteed that the bug(s) are found.

7.4 Research Question 4

The fourth and last research question is: "What is the relationship between the duration of the time budget and the effectiveness of a selected test set, and how can the optimal time budget be determined for achieving the best results?"

As with research question three, this was addressed in the evaluation chapter. The commit size analysis showed that for the case study project, up to 45 files can be changed, and a single bug can still be found in 80 percent of cases. This indicates that the effectiveness of the developed approach can be comparable to running a full test suite. However, because of the effort required to validate this, the case study project was the only one tested. More data is required to draw a final conclusion on the effectiveness.

The collected data shows that the duration of the ideal time budget is highly dependent on factors like the number of changes, whether every bug should be found, and the duration of each test case. Regardless a conclusion from the data can be made which is that the effectiveness does not peak at a specific time budget and that a higher time budget yields generally better results than a lower one.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 8

Conclusion

This work tried to investigate whether a time-constrained approach to software testing is possible. The developed method utilizes a genetic algorithm and per-test-file-based coverage. It outperformed the Random Testing benchmark. The experiments conducted during the evaluation indicate that finding bugs in a reduced amount of time is possible. However, the analysis of the commit size suggests that it cannot be guaranteed that all bugs are found.

8.1 Threats to Validity

While the work done for this thesis was done with as much rigor as possible, there are still a few points that could threaten the validity of the presented results.

The first issue is that the approach was only tested on a single case study project. The reason for this is the enormous amount of resources required for analyzing the approach, especially on the CI side. For the data presented in the evaluation chapter, over 14800 pipelines were executed, totaling over 40.000 pipeline minutes, over 28 days. Therefore, the decision was to keep this for future work.

The second issue is that no statistical analysis was conducted. There are two main reasons why this is the case. The first one is that the evaluation was conducted only with artificially introduced bugs. While this ensures better comparability, it also diminishes the significance of the data. Ideally, the data for this analysis would be collected with actual commits, where the commit is tested with different time budgets for the developed approach and the RT approach. However, the required resources to test multiple commits are enormous.

The third issue is that no actual commits were used for evaluation. The reason for this is better comparability of artificially introduced bugs. However, ideally, to collect this data,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the approach would have to be utilized in a production environment for some time in order to generate enough data. This was not feasible for this work.

8.2 Future Work

The following things are considered to be of interest for future work:

- Implement a Strict Time Budget: Ideally, to restrict the execution time to a specified time, a custom test runner would need to be implemented.
- Different Projects: It would be interesting to apply the approach to different case study projects in order to generate more data to prove the effectiveness of the approach.
- Allow More Commits: As stated in the method design chapter (chapter 5), currently, only 1000 commits are considered for changes; it would be ideal if this restriction is removed.
- Deploy to Production: As stated in the threats to validity, deployment to a production environment would bring additional data that could underscore the feasibility of the developed approach.
- Different Selection Methods: It would be interesting to see how other methods to select the test cases like Ant Colony Optimization mentioned in chapter3.2 compare to the genetic algorithm.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

2.1 Overview of CI/CD pipeline taken from [29] 11
2.2 General outline of evolutionary algorithm by Spears [78] 13
2.3 GA vs steady state GA [85] 15
2.4 Genetic algorithm pseudo-code [85] 15
2.5 Steady state genetic algorithm pseudo-code [85] 16

3.1 Summary of the coverage criteria effectiveness in five subjects of study from [43] 18
3.2 Percent of errors found per project [43] 19
3.3 Average percent of errors found per metric [43] 20
3.4 Evaluation of first company [64] 25
3.5 Evaluation of second company [64] 26
3.6 Feature diagram notations [6] 27
3.7 A feature diagram and its grammar [6] 27
3.8 Overall overview of the proposed approach for test case prioritization. [2] 27
3.9 Example of an ant colony system for Traveling Salesman Problem [22] . . 29
3.10 Graph for Coverage-Based Ant Colony System (CB-ACS) [51] 29
3.11 Overall overview of the proposed approach for test case prioritization [51] 30
3.12 Overview of the approach presented by Rajsingh, Kumar, and Srinivasan [68] 32
3.13 FYSSOA pseudo-code [68] 33

4.1 Design cycle (Wieringa) [86] 36

5.1 Iteration 1: Backtracking architecture design 43
5.2 Evaluation of iteration 1: Boxplot backtracking execution times from table 5.1 47
5.3 Iteration 2: Genetic algorithm architecture design 50
5.4 Iteration 2: Genetic algorithm architecture design 51
5.5 Iteration 2: Genetic algorithm architecture design 52
5.6 Evaluation of iteration 2: Boxplot of execution times from table 5.2 . . . 56
5.7 Evaluation of iteration 2: Boxplot of execution times from table 5.4 . . . 57
5.8 Iteration 3: Entity relation diagram of database 59
5.9 Iteration 3: Postgres adapter design 60
5.10 Iteration 5: Flow diagram of developed method 64

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.11 Evaluation of iteration 5: Boxplot of execution times from table 5.7 . . . 69
5.12 Iteration 5: Flow diagram of developed method 71
5.13 Iteration 5: PostgresAdaper design 72
5.14 Evaluation of iteration 6: Boxplot of execution times from table 5.8 . . . 77
5.15 Iteration 7: Flow Diagram of RT approach 79
5.16 Evaluation of iteration 8: RT approach: Boxplot diagram of execution times from table 5.10 80
5.17 Iteration 8: RT approach: Improved flow diagram 82
5.18 Evaluation of iteration 8: Boxplot of execution times from table 5.11 . . . 83

6.1 Developed method: Percentage of pipelines finding the introduced bug with x additional files in commit 86
6.2 Random testing: Percentage of pipelines finding the introduced bug with x additional files in commit 88
6.3 Improved algorithm: Percentage of pipelines finding at least one bug with one bug introduced plus x additional files in commit 89
6.4 Improved algorithm: Percentage of pipelines finding at least one bug with two bugs introduced plus x additional files in commit 90
6.5 Developed method: Percentage of pipelines finding at least one bug with three bugs introduced plus x additional files in commit 90
6.6 Developed method: Percentage of pipelines finding at least one bug with five bugs introduced plus x additional files in commit 91
6.7 Developed method: Number of bugs found on average with one bug introduced plus x additional files in commit 91
6.8 Developed method: Number of bugs found on average with two bugs introduced plus x additional files in commit 92
6.9 Developed method: Number of bugs found on average with three bugs introduced plus x additional files in commit 92
6.10 Developed method: Number of bugs found on average with five bugs introduced plus x additional files in commit 93

7.1 Pre-loop phase of the developed method 97
7.2 Loop phase of the developed method 98
7.3 Test selection and execution of the developed method 98

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

5.1 Evaluation of iteration 1: Execution times of backtracking run 46
5.2 Evaluation of iteration 2: Execution times of different time budgets and 5 bugs introduced. (100 iterations, population: 50) 55
5.3 Evaluation of iteration 2: Number of bugs found with different time budgets and five bugs introduced. (100 Iterations, Population: 50) 56
5.4 Evaluation of iteration 2: Execution times of different time budgets and five bugs introduced (500 iterations, population: 100) 57
5.5 Evaluation of iteration 2: Number of bugs found with different time budgets and five bugs introduced. (500 Iterations, Population: 100) 58
5.6 Evaluation of iteration 5: Number of bugs found with different time budgets and five bugs introduced 68
5.7 Evaluation of iteration 5: Execution times of different time budgets and five bugs introduced 69
5.8 Evaluation of iteration 6: Execution times of different time budgets and five bugs introduced 77
5.9 Evaluation of iteration 6: Number of bugs found with different time budgets and five bugs introduced 78
5.10 Evaluation of iteration 7: RT approach: Evaluation of keeping time budget 80
5.11 Evaluation of iteration 8: RT Approach: Evaluation of keeping time budget 83

6.1 Developed method: Percentage of pipelines finding the introduced bug with x additional files in commit 86
6.2 RT approach: Percentage of pipelines finding the introduced bug with x additional files in commit 88

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

5.1 Iteration 1: Backtracking algorithm implementation 45
5.2 Initial basic fitness 53
5.3 Genetic algorithm orchestration script 54
5.4 Execution algorithm with three runs part one 66
5.5 Execution algorithm with three runs part two 67
5.6 Fitness functions of new genetic algorithm 74
5.7 Crossover function of new genetic algorithm 75
5.8 Combine coverage helper function used by crossover 75
5.9 Finish function of new genetic algorithm 76

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

ACB Addition Coverage-Based Heuristic. 28

ACO Ant Colony Optimization. 28, 102

ACS Ant Colony System. 28

ART Adaptive Random Testing. 21

CB-ACS Coverage-Based Ant Colony System. 28, 29, 41, 42, 47, 103

CI Continuous Integration. ix, xi, 1–4, 10, 11, 40, 41, 58, 101

CI/CD Continuous Integration/Continuous Delivery. 7, 10, 11

CPS Cyber-Physical Systems. 26

CSV Comma-Separated Values. 48

ED-EPCA Elliptical Distributions-centric Emperor Penguins Colony Algorithm. 31, 33

EP Extreme Programming. 1

FSCS Fixed-Size Candidate Set. 21

FY-SSOA Fishers Yates Shuffled Shepherd Optimization Algorithm. 31–33

GUI Graphical User Interface. 9

MC/DC Modified Condition-Decision Coverage. 18

MVP Minimum Viable Product. 39

NCD Normalized Compression Distance. 23, 24

NID Normalized Information Distance. 24

NPM Node Package Manager. 42

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

OOP Object Oriented Programming. 48

RT Random Testing. 4, 20, 21, 31, 40, 78, 80, 81, 83, 85, 101

SCM Source Code Management. 2, 10, 11, 35, 95, 96

SUT System Under Test. 9, 32

TSP Traveling Salesman Problem. 28, 29, 103

UI User Interface. 9, 40, 61

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

[1] Andrea Adamoli et al. "Automated GUI performance testing". eng. In: *Software quality journal* 19.4 (2011), pp. 801–839. ISSN: 0963-9314. DOI: 10.1007/s11219-011-9135-x.

[2] Aitor Arrieta et al. "Search-Based test case prioritization for simulation-Based testing of cyber-Physical system product lines". eng. In: *The Journal of systems and software* 149 (2019), pp. 1–34. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.09.055.

[3] Aitor Arrieta et al. "Some Seeds are Strong: Seeding Strategies for Search-based Test Case Selection". eng. In: *ACM transactions on software engineering and methodology* (2022). ISSN: 1049-331X. DOI: 10.1145/3532182.

[4] A.N. Arslan and O. Egecioglu. "An efficient uniform-cost normalized edit distance algorithm". eng. In: *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No.PR00268)*. IEEE, 1999, pp. 8–15. ISBN: 0769502687. DOI: 10.1109/SPIRE.1999.796572.

[5] Abdullah N Arslan and Omer Egecioglu. *Efficient Algorithms For Normalized Edit Distance*. Tech. rep.

[6] Don Batory. "Feature models, grammars, and propositional formulas". eng. In: *Lecture notes in computer science*. 1ère éd. New York, NY: Springer, 2005, pp. 7–20. ISBN: 9783540289364. DOI: 10.1007/11554844_3.

[7] C.H. Bennett et al. "Information distance". eng. In: *IEEE transactions on information theory* 44.4 (1998), pp. 1407–1423. ISSN: 0018-9448. DOI: 10.1109/18.681318.

[8] M. Bernhart et al. "Applying Continuous Code Reviews in Airport Operations Software". eng. In: *2012 12th International Conference on Quality Software*. IEEE, 2012, pp. 214–219. ISBN: 146732857X. DOI: 10.1109/QSIC.2012.61.

[9] Yi Bian et al. "Epistasis Based ACO for Regression Test Case Prioritization". eng. In: *IEEE transactions on emerging topics in computational intelligence* 1.3 (2017), pp. 213–223. ISSN: 2471-285X. DOI: 10.1109/TETCI.2017.2699228.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[10] Emanuela G Cartaxo, Francisco G O Neto, and Patrícia D L Machado. "Automated Test Case Selection Based on a Similarity Function". In: *Informatik 2007 – Informatik trifft Logistik – Band 2*. Bonn: Gesellschaft für Informatik e. V., 2007, pp. 381–386. ISBN: 978-3-88579-206-1.

[11] Emanuela G. Cartaxo, Patrícia D. L. Machado, and Francisco G. Oliveira Neto. "On the use of a similarity function for test case selection in the context of model-based testing". eng. In: *Software testing, verification & reliability* 21.2 (2011), pp. 75–100. ISSN: 0960-0833. DOI: 10.1002/stvr.413.

[12] T. Y. Chen, De Hao Huang, and Zhi Quan Zhou. "Adaptive Random Testing Through Iterative Partitioning". eng. In: *Ada-Europe*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 155–166. ISBN: 9783540346630. DOI: 10.1007/11767077_13.

[13] T.Y. Chen, T.H. Tse, and Y.T. Yu. "Proportional sampling strategy: a compendium and some insights". eng. In: *The Journal of systems and software* 58.1 (2001), pp. 65–81. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(01)00028-0.

[14] Tsong Chen and Robert Merkel. "An upper bound on software testing effectiveness". eng. In: *ACM transactions on software engineering and methodology* 17.3 (2008), pp. 1–27. ISSN: 1049-331X. DOI: 10.1145/1363102.1363107.

[15] Tsong Yueh Chen, Fei-Ching Kuo, and Zhi Quan Zhou. "On favourable conditions for adaptive random testing". In: *International Journal of Software Engineering and Knowledge Engineering* 17.06 (2007), pp. 805–825.

[16] Tsong Yueh Chen et al. "Adaptive Random Testing: The ART of test case diversity". eng. In: *The Journal of systems and software* 83.1 (2010), pp. 60–66. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.02.022.

[17] R. Cilibrasi and P.M.B. Vitanyi. "Clustering by compression". eng. In: *IEEE Transactions on Information Theory* 51.4 (2005), pp. 1523–1545. ISSN: 0018-9448. DOI: 10.1109/TIT.2005.844059.

[18] Ilinca Ciupa et al. "ARTOO: adaptive random testing for object-oriented software". eng. In: *International Conference on Software Engineering 2008*. Vol. 2008. 24. ACM, 2008, pp. 71–80. ISBN: 1605580791. DOI: 10.1145/1368088.1368099.

[19] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. "On the journey to continuous deployment: Technical and social challenges along the way". eng. In: *Information and software technology* 57.1 (2015), pp. 21–31. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2014.07.009.

[20] A.C. Coulter. "Graybox software testing methodology: embedded software testing technique". eng. In: *Gateway to the New Millennium. 18th Digital Avionics Systems Conference. Proceedings (Cat. No.99CH37033)*. Vol. 2. IEEE, 1999, pp. 5–10. ISBN: 0780357493. DOI: 10.1109/DASC.1999.822089.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[21] Ana Emília Victor Barbosa Coutinho, Emanuela Gadelha Cartaxo, and Patrícia Duarte de Lima Machado. "Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing". eng. In: *Software quality journal* 24.2 (2016), pp. 407–445. ISSN: 0963-9314. DOI: 10.1007/s11219-014-9265-z.

[22] M. Dorigo and L.M. Gambardella. "Ant colony system: a cooperative learning approach to the traveling salesman problem". eng. In: *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 53–66. ISSN: 1089-778X. DOI: 10.1109/4235.585892.

[23] Paul M. Duvall, Andrew Glover, and Steve Matyas. *Continuous integration : improving software quality and reducing risk*. eng. 1st edition. Addison-Wesley signature series. [Place of publication not identified]: Addison Wesley, 2007. ISBN: 0321518438.

[24] Emelie Engström, Per Runeson, and Mats Skoglund. "A systematic review on regression test selection techniques". eng. In: *Information and software technology* 52.1 (2010), pp. 14–30. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.07.001.

[25] Emelie Engström, Mats Skoglund, and Per Runeson. "Empirical evaluations of regression test selection techniques: a systematic review". eng. In: *ESEM'08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2008, pp. 22–31. ISBN: 9781595939715. DOI: 10.1145/1414004.1414011.

[26] Fabio Farzat and Márcio de O. Barros. "Unit Test Case Selection to Evaluate Changes in Critical Time". eng. In: *INFOR. Information systems and operational research* 50.4 (2012), pp. 163–174. ISSN: 0315-5986. DOI: 10.3138/infor.50.4.163.

[27] Robert Feldt et al. "Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics". In: *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. 2008, pp. 178–186. DOI: 10.1109/ICSTW.2008.36.

[28] Robert Feldt et al. "Test Set Diameter: Quantifying the Diversity of Sets of Test Cases". In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016, pp. 223–233. DOI: 10.1109/ICST.2016.33.

[29] Brian Fitzgerald and Klaas-Jan Stol. "Continuous software engineering: A roadmap and agenda". eng. In: *The Journal of systems and software* 123 (2017), pp. 176–189. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.06.063.

[30] Todd Graves et al. "An empirical study of regression test selection techniques". eng. In: *ACM transactions on software engineering and methodology* 10.2 (2001), pp. 184–208. ISSN: 1049-331X. DOI: 10.1145/367008.367020.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[31] M.J. Harrold and M.L. Soffa. "Selecting and using data for integration testing". eng. In: *IEEE software* 8.2 (1991), pp. 58–65. ISSN: 0740-7459. DOI: 10.1109/52.73750.

[32] Mary Jean Harrold. "Testing evolving software". eng. In: *The Journal of systems and software* 47.2 (1999), pp. 173–181. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(99)00037-0.

[33] Mary Jean Harrold et al. "Regression test selection for Java software". eng. In: *ACM SIGPLAN Notices* 36.11 (2001), pp. 312–326. ISSN: 0362-1340. DOI: 10.1145/504311.504305.

[34] Randy L Haupt and S. E Haupt. *Practical genetic algorithms*. eng. Ed. by S. E Haupt. 2nd ed. Hoboken, N.J.: John Wiley, 2004. ISBN: 1280542128.

[35] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. "Achieving scalable model-based testing through test case diversity". eng. In: *ACM transactions on software engineering and methodology* 22.1 (2013), pp. 1–42. ISSN: 1049-331X. DOI: 10.1145/2430536.2430540.

[36] Hadi Hemmati et al. "Prioritizing manual test cases in rapid release environments". eng. In: *Software testing, verification & reliability* 27.6 (2017), e1609–n/a. ISSN: 0960-0833. DOI: 10.1002/stvr.1609.

[37] Christopher Henard et al. "Comparing White-Box and Black-Box Test Prioritization". eng. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 523–534. ISBN: 9781450339001. DOI: 10.1145/2884781.2884791.

[38] Marko Ivankovi et al. "Code coverage at Google". In: *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Inc, Aug. 2019, pp. 955–963. ISBN: 9781450355728. DOI: 10.1145/3338906.3340459.

[39] Paul Jaccard. "The Distribution of the Flora in the Alpine Zone". eng. In: *The New phytologist* 11.2 (1912), pp. 37–50. ISSN: 0028-646X. DOI: 10.1111/j.1469-8137.1912.tb05611.x.

[40] H. Jaygarl, C.K. Chang, and Sunghun Kim. "Practical Extensions of a Randomized Testing Tool". eng. In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 1. IEEE, 2009, pp. 148–153. ISBN: 076953726X. DOI: 10.1109/COMPSAC.2009.29.

[41] Rolf Johansson. "On Case Study Methodology". eng. In: *Open house international* 32.3 (2007), pp. 48–54. ISSN: 0168-2601. DOI: 10.1108/OHI-03-2007-B0006.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[42] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval ...* Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boton: Springer US, 1972, pp. 85–103.

[43] Rafaqut Kazmi et al. “Effective Regression Test Case Selection”. eng. In: *ACM computing surveys* 50.2 (2018), pp. 1–32. ISSN: 0360-0300. DOI: 10.1145/3057269.

[44] Zubair Khaliq, Sheikh Umar Farooq, and Dawood Ashraf Khan. “Transformers for GUI Testing: A Plausible Solution to Automated Test Case Generation and Flaky Tests”. eng. In: *Computer (Long Beach, Calif.)* 55.3 (2022), pp. 64–73. ISSN: 0018-9162. DOI: 10.1109/MC.2021.3136791.

[45] Onur Kilincceker et al. “Model-Based Ideal Testing of GUI Programs-Approach and Case Studies”. eng. In: *IEEE access* 9 (2021), pp. 68966–68984. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3077518.

[46] F.J. Lacoste. “Killing the Gatekeeper: Introducing a Continuous Integration System”. eng. In: *2009 Agile Conference*. IEEE, 2009, pp. 387–392. ISBN: 9780769537689. DOI: 10.1109/AGILE.2009.35.

[47] Janusz Laski and William Stanley. “Is There a Bug in the Program? Structural Program Testing”. eng. In: *Software Verification and Analysis*. London: Springer London, 2009, pp. 173–202. ISBN: 9781848822399. DOI: 10.1007/978-1-84882-240-5_8.

[48] Yves Ledru et al. “Prioritizing test cases with string distances”. eng. In: *Automated software engineering* 19.1 (2012), pp. 65–95. ISSN: 0928-8910. DOI: 10.1007/s10515-011-0093-0.

[49] Ming Li and Vitányi Paul. *An Introduction to Kolmogorov Complexity and Its Applications*. eng. 4th ed. 2019. Texts in Computer Science. Cham: Springer Nature, 2019. ISBN: 3030112985. DOI: 10.1007/978-3-030-11298-1.

[50] Doina Logofătu. *Grundlegende Algorithmen mit Java : Vom Algorithmus zum fertigen Programm — Lern- und Arbeitsbuch für Informatiker und Mathematiker*. Ed. by Doina Logofătu. Wiesbaden: Friedr. Vieweg & Sohn Verlag | GWV Fachverlage GmbH, Wiesbaden, 2008. ISBN: 3834894338. DOI: 10.1007/978-3-8348-9433-5.

[51] Chengyu Lu et al. “Ant Colony System With Sorting-Based Local Search for Coverage-Based Test Case Prioritization”. eng. In: *IEEE transactions on reliability* 69.3 (2020), pp. 1004–1020. ISSN: 0018-9529. DOI: 10.1109/TR.2019.2930358.

[52] A. Marzal and E. Vidal. “Computation of normalized edit distance and applications”. eng. In: *IEEE transactions on pattern analysis and machine intelligence* 15.9 (1993), pp. 926–932. ISSN: 0162-8828. DOI: 10.1109/34.232078.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[53] Philipp Mayring. *Qualitative Inhaltsanalyse : Grundlagen und Techniken*. ger. Ed. by Julius Beltz GmbH & Co. KG and Verlag. 13., überarbeitet... Weinheim: Beltz, 2022. ISBN: 3407258984.

[54] Bertrand Meyer and Martin Nordio. "Is Branch Coverage a Good Measure of Testing Effectiveness?" eng. In: *Empirical Software Engineering and Verification*. Vol. 7007. Lecture Notes in Computer Science. Germany: Springer Berlin / Heidelberg, 2012, pp. 194–212. ISBN: 3642252303. DOI: 10.1007/978-3-642-25231-0_5.

[55] Ming Li et al. "The similarity metric". eng. In: *IEEE transactions on information theory* 50.12 (2004), pp. 3250–3264. ISSN: 0018-9448. DOI: 10.1109/TIT.2004.838101.

[56] Glenford J Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*. eng. 3. Aufl. Hoboken: Wiley, 2011. ISBN: 9781118031964. DOI: 10.1002/9781119202486.

[57] Glenford J. Myers et al. *The art of software testing*. eng. Ed. by Tom Badgett, Todd M. Thomas, and Corey Sandler. 2nd ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2004. ISBN: 1280346167.

[58] Daniel Di Nardo et al. "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system". eng. In: *Software testing, verification & reliability* 25.4 (2015), pp. 371–396. ISSN: 0960-0833. DOI: 10.1002/stvr.1572.

[59] Michel Nass, Emil Alégroth, and Robert Feldt. "Why many challenges with GUI test automation (will) remain". eng. In: *Information and software technology* 138 (2021), p. 106625. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106625.

[60] Steve Neely and Steve Stolt. "Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)". In: *2013 Agile Conference*. 2013, pp. 121–128. DOI: 10.1109/AGILE.2013.17.

[61] Thomas E. Nisonger. "The "80/20 Rule" and Core Journals". eng. In: *The Serials librarian* 55.1-2 (2008), pp. 62–84. ISSN: 0361-526X. DOI: 10.1080/03615260801970774.

[62] Tanzeem Bin Noor and Hadi Hemmati. "A similarity-based approach for test case prioritization using historical failure data". In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 2015, pp. 58–68. DOI: 10.1109/ISSRE.2015.7381799.

[63] Gerard O'Regan. *Introduction to Software Quality*. eng. Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2014. ISBN: 3319061062. DOI: 10.1007/978-3-319-06106-1.

[64] Francisco de Oliveira Neto et al. "Improving continuous integration with similarity-based test case selection". eng. In: *2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*. AST '18. ACM, 2018, pp. 39–45. ISBN: 1450357431. DOI: 10.1145/3194733.3194744.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[65] Francisco G. de Oliveira Neto, Richard Torkar, and Patrícia D.L. Machado. "Full modification coverage through automatic similarity-based test case selection". eng. In: *Information and software technology* 80 (2016), pp. 124–137. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.08.008.

[66] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. "Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software". In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. 2012, pp. 392–399. DOI: 10.1109/SEAA.2012.54.

[67] Dipesh Pradhan et al. "Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1085–1092. ISBN: 9781450342063. DOI: 10.1145/2908812.2908850. URL: https://doi.org/10.1145/2908812.2908850.

[68] J. Paul Rajasingh, P. Senthil Kumar, and S. Srinivasan. "Efficient Fault Detection by Test Case Prioritization via Test Case Selection". eng. In: *Journal of electronic testing* 39.5-6 (2023), pp. 659–677. ISSN: 0923-8174. DOI: 10.1007/s10836-023-06086-3.

[69] R. Owen ROGERS. "Scaling continuous integration". eng. In: *Lecture notes in computer science*. Berlin: Springer, 2004, pp. 68–76. ISBN: 9783540221371. DOI: 10.1007/978-3-540-24853-8_8.

[70] G. Rothermel and M.J. Harrold. "Analyzing regression test selection techniques". eng. In: *IEEE transactions on software engineering* 22.8 (1996), pp. 529–551. ISSN: 0098-5589. DOI: 10.1109/32.536955.

[71] G. Rothermel and M.J. Harrold. "Empirical studies of a safe regression test selection technique". eng. In: *IEEE transactions on software engineering* 24.6 (1998), pp. 401–419. ISSN: 0098-5589. DOI: 10.1109/32.689399.

[72] G. Rothermel et al. "Prioritizing test cases for regression testing". eng. In: *IEEE transactions on software engineering* 27.10 (2001), pp. 929–948. ISSN: 0098-5589. DOI: 10.1109/32.962562.

[73] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. "Regression test selection for C++ software". eng. In: *Software testing, verification & reliability* 10.2 (2000), pp. 77–109. ISSN: 0960-0833. DOI: 10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E.

[74] Seojin Kim et al. "Automated Continuous Integration of Component-Based Software: An Industrial Experience". eng. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 423–426. ISBN: 142442187X. DOI: 10.1109/ASE.2008.64.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

[75] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices”. eng. In: *IEEE access* 5 (2017), pp. 3909–3943. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2685629.

[76] Arnaldo Marulitua Sinaga. “BRANCH COVERAGE BASED TEST CASE PRIORITIZATION”. In: 10.3 (2015). ISSN: 1819-6608. URL: www.arpnjournals.com.

[77] Yogesh Singh. *Software testing*. eng. Cambridge: Cambridge University Press, 2011. ISBN: 1139196189. DOI: 10.1017/CBO9781139196185.

[78] William M Spears. *Evolutionary algorithms : the role of mutation and recombination ; with 23 tables*. eng. Natural computing series. Berlin [u.a.]: Springer, 2000. ISBN: 3540669507.

[79] S. Stolberg. “Enabling Agile Testing through Continuous Integration”. eng. In: *2009 Agile Conference*. IEEE, 2009, pp. 369–374. ISBN: 9780769537689. DOI: 10.1109/AGILE.2009.16.

[80] Ian Sutton. *Process risk and reliability management*. eng. 2nd ed. Oxford: GPP, 2015. ISBN: 0128017961.

[81] Nguyen Ngoc Thach, Francisco Zapata, and Olga Kosheleva. “When to stop testing software: economic approach”. eng. In: *Soft computing (Berlin, Germany)* 25.12 (2021), pp. 7985–7989. ISSN: 1432-7643. DOI: 10.1007/s00500-021-05818-x.

[82] Paulo José Azevedo Vianna Ferreira and Márcio de Oliveira Barros. “Traceability between function point and source code”. In: *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*. TEFSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 10–16. ISBN: 9781450305891. DOI: 10.1145/1987856.1987860. URL: https://doi.org/10.1145/1987856.1987860.

[83] E. Vidal, A. Marzal, and P. Aibar. “Fast computation of normalized edit distances”. eng. In: *IEEE transactions on pattern analysis and machine intelligence* 17.9 (1995), pp. 899–902. ISSN: 0162-8828. DOI: 10.1109/34.406656.

[84] Neil Walkinshaw. *Software Quality Assurance*. eng. 1st ed. Undergraduate Topics in Computer Science. Cham: Springer Nature, 2017. ISBN: 3319648225. DOI: 10.1007/978-3-319-64822-4.

[85] Karsten Weicker. *Evolutionäre Algorithmen*. ger. 2., überarb. u. e... Leitfäden der Informatik. Wiesbaden: Teubner, 2007. ISBN: 3835102192.

[86] Roel J Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. eng. 2014 edition. Berlin, Heidelberg: Springer Nature, 2014. ISBN: 9783662438398. DOI: 10.1007/978-3-662-43839-8.

[87] Thomas Wilde and Thomas Hess. *Methodenspektrum der Wirtschaftsinformatik: Überblick und Portfoliobildung*. Tech. rep. URL: http://www.wim.bwl.uni-muenchen.de.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[88] D.H. Wolpert and W.G. Macready. "No free lunch theorems for optimization". eng. In: *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893.

[89] S. Yoo and M. Harman. "Regression testing minimization, selection and prioritization: a survey". eng. In: *Software testing, verification & reliability* 22.2 (2012), pp. 67–120. ISSN: 0960-0833. DOI: 10.1002/stv.430.

[90] Li Yujian and Liu Bo. "A Normalized Levenshtein Distance Metric". eng. In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1091–1095. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2007.1078.

[91] Zhi Quan Zhou. "Using Coverage Information to Guide Test Case Selection in Adaptive Random Testing". eng. In: *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*. IEEE, 2010, pp. 208–213. ISBN: 1424480892. DOI: 10.1109/COMPSACW.2010.43.

[92] Xingni Zhou, Lei Feng, and Qiguang Miao. *Programming in C : Volume 1: Basic Data Structures and Program Statements*. eng. Ed. by Lei Feng and Qiguang Miao. De Gruyter Textbook. Berlin Boston: De Gruyter, 2020. ISBN: 3110692325.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.