Université de Lille

Faculté des Sciences et Technologies

Selection and Prioritization of Test Cases for Software Testing Optimization

Samuel DUBUISSON

Master Informatique

Master in Computer Science

Département d'Informatique

Faculté des Sciences et Technologies

March, 2025

This report partially satisfies the requirements defined for the Project/Internship course, in the 3^{rd} year, of the Master in Computer Science.

Candidate: Samuel DUBUISSON, No. 41935757,
`samuel.dubuisson.etu@univ-lille.fr`

Scientific Guidance: Clément QUINTON, `clement.quinton@univ-lille.fr`

Département d'Informatique
Faculté des Sciences et Technologies
Campus Cité Scientifique, Bât. M3 extension, 59655 Villeneuve-d'Ascq

March, 2025

Abstract

To ensure the proper functioning of an application, the development team often equips it with a robust test suite. These tests allow, among other things, verifying that the application's behavior still complies with its specifications after new features are added: the tests specifically responsible for this aspect are called regression tests. It is common practice to integrate them into a continuous integration loop to execute them with each new code addition. However, this existing test base grows as the project becomes larger, and execution times can quickly become unacceptable, either for the development workflow or due to the strain on infrastructure. Thus, it becomes crucial to develop strategies to accelerate this process. The first strategy is the selection of tests to execute: this involves avoiding the execution of unnecessary tests or those that do not provide new information and automating this process. For example, if it can be predicted that a certain code addition does not influence the result of a particular test, then it is relevant not to execute it. The second method explored here, which is currently the most popular, is the automatic scheduling of tests. This consists of executing tests in an order that allows the most information to be obtained as quickly as possible (the challenge being to determine which information is relevant and how to access it as rapidly as possible). This approach allows, for instance, providing faster feedback to developers or quickly stopping the process if an initial error is detected, thus relieving the infrastructure. The goal here will be to present and precisely define these two processes and their evaluation criteria. Then, we will focus on the various implementations of these strategies, highlighting their strengths, weaknesses, and use cases. Finally, we will examine the current industrial applications of these strategies and the challenges that remain.

Keywords: Test case selection, Test suite prioritization, Regression testing, Continuous integration

Résumé

Afin d'assurer le bon fonctionnement d'une application, l'équipe de développeurs l'accompagne souvent d'une suite robuste de tests. Ces tests sont essentiels au bon fonctionnement de l'application et permettent, entre autres, de vérifier que le comportement de l'application est toujours conforme à son cahier des charges après l'ajout de nouvelles fonctionnalités : les tests précisément en charge de cette partie sont appelés tests de régression. Il est courant de les intégrer dans une boucle d'intégration continue, afin de les exécuter à chaque nouvel ajout de code. Or, cette base de tests existants augmente à mesure que le projet devient plus important, et les temps d'exécution peuvent rapidement devenir inacceptables, que ce soit au niveau du temps d'attente des développeurs ou de la demande de calcul envoyée aux infrastructures. Ainsi, il devient crucial de développer des stratégies pour accélérer ce processus. La première stratégie est la sélection des tests à exécuter : il s'agit d'éviter d'exécuter les tests non nécessaires ou qui n'apportent pas de nouvelles informations, et d'automatiser ce processus. Par exemple, si l'on peut prévoir que tel ajout de code n'influence pas le résultat de tel test, alors il est pertinent de ne pas l'exécuter. La seconde stratégie étudiée ici, qui est la plus en vogue actuellement, est l'ordonnancement automatique des tests. Il s'agit d'exécuter les tests dans un ordre qui permet d'obtenir le plus d'informations le plus rapidement possible (tout l'enjeu étant de déterminer quelles informations sont pertinentes et comment les obtenir le plus rapidement possible). Cela permet, par exemple, d'offrir au développeur un retour plus rapide ou de stopper le processus rapidement si une première erreur est détectée, soulageant ainsi l'infrastructure. Le but ici sera donc de présenter et de définir précisément ces deux procédés ainsi que leurs critères d'évaluation. Ensuite, nous nous intéresserons aux différentes implémentations de ces stratégies, en précisant leurs atouts, inconvénients et cas d'usage. Enfin, nous examinerons les applications actuelles de ces stratégies dans l'industrie et les défis à relever.

Mots-clés: Sélections de tests, Ordonnancement de tests, Tests de régression, Intégration continue

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence

APBC Average Percentage of Branch Coverage

APCC Average Percentage of Condition Coverage

APEC Analysis - Planning and Preparation - Execution - Closure

APFD Average Percentage of Faults Detected

APFDc Average Percentage of Faults Detected with Cost

APLC Average Percentage of Loop Coverage

APSC Average Percentage of Statement Coverage

CI Continuous Integration

CORE Computing Research Education Association of Australasia

DevOps Development Operations

DFG DataFlow Graph

FEP Fault-Exposing Potential

GA Genetic Algorithms

HyRTS Hybrid Regression Test Selection

IT Information Technology

MBS Model-Base Selection

ML Machine Learning

PTR Problem Tracking Report

RTS Regression Test Selection

TCP Test Case Prioritization

TCS Test Case Selection

UML Unified Modeling Language

Chapter 1

Introduction

1.1 Background

The nature of software development is a subject of ongoing debate. Is it purely an engineering discipline, a meticulous craft, or even an art form? The truth likely lies in the intersection of all three. Software development encompasses structured engineering principles, the hands-on craftsmanship of writing code, and the creative ingenuity required to design elegant solutions. At its core, software engineering represents the application of engineering methodologies to software creation. This involves systematic processes, structured design principles, and rigorous quality control measures to ensure reliability and efficiency. One of the fundamental pillars of software engineering is ensuring that the software functions correctly, and this is achieved through rigorous testing.

Software testing is a fundamental aspect of software development, ensuring that applications function correctly, efficiently, and securely. Hooda and Chiilar [5] define software testing as "the main activity of evaluating and executing software with a view to find out errors". Beyond simply detecting defects, testing verifies that a system meets its requirements and performs as expected under various conditions. As software systems grow increasingly complex, the role of testing continues to expand, ensuring stability and trustworthiness. Given the fast-paced evolution of technology, testing is no longer an isolated phase but an integral part of modern development workflows, particularly with the rise of continuous integration and deployment practices. The test suite must live along the program development.

During the development, tests are added but the old ones should be progressively modified or even deleted for the obsolete ones.

1.1.1 Software Testing Process

The software testing process follows a structured approach, commonly summarized by the Analysis - Planning and Preparation - Execution - Closure (APEC) model [5]. It begins with an analysis of functional and non-functional requirements to define clear objectives. This is followed by planning and preparation, where test cases, test data, and test environments are set up. The execution phase involves running the tests and analyzing results, while the closure phase assesses overall success, identifies areas for improvement, and ensures that all objectives have been met.

Software testing can be categorized into several key types [5]. Functional testing ensures that the software behaves as expected based on its specifications. This includes unit testing, which verifies individual components, integration testing, which checks interactions between modules, and acceptance testing, where end users validate the software's behavior. Performance testing evaluates how well a system performs under different conditions, measuring factors such as response time, load handling, and reliability. Techniques such as stress testing, soak testing, and spike testing help assess system stability. Security testing focuses on identifying vulnerabilities and ensuring that data and resources are protected against threats. This includes verifying authentication mechanisms, encryption methods, and access controls while addressing common security flaws.

1.1.2 Regression Testing

Regression testing is not a testing type, as it groups all of the different other testing types. It's more like a testing process. Leung and White [6] are the first ones that clearly introduce this concept. Regression testing "involves testing the modified program with some test cases in order to re-establish our confidence that the program will perform according to the (possibly modified) specification". The confidence is established only if the program follows a high quality testing plan.

Moreover, Regression Testing can be divided into two different groups named Progressive Regression Testing and Corrective Regression Testing. Progressive Regression Testing relates to major changes when specifications are changed. It involves major modifications in the program (adding or deleting modules) and implies that fewer test cases can be reused. On the other hand, Corrective Regression Testing relates to minor changes: The specification remains the same and is performed during development and corrective maintenance and most of the test cases can be reused.

Software testing has become mandatory nowadays to develop high-quality products. As we have seen, it enhances robustness, quality, security and performances.

Moreover, it's fully integrated in the trendy Development Operations (DevOps) software process [7], ensuring automatic deployments of valid softwares and allows quick feedbacks for the development team.

1.2 Problem Statement

The complexity and scale of Information Technology (IT) projects are steadily increasing, and as projects grow in size, they become more susceptible to bugs, unexpected issues, security, performance or functional issues. To solve these issues, we have presented the importance of testing and Regression testing. However while essential, regression testing can introduce significant performance and resource-related challenges. As the test suite grows along with the project, running all tests frequently becomes time-consuming and costly. These challenges demand efficient approaches to regression testing, particularly in terms of optimizing which tests to run and in what order.

Running all the tests for each modification is not efficient and should not be considered. To underline that with numbers, a standard video conference application test suite could take 2 days to run entirely [8], and large companies such as Amazon experiment up to 136,000 system deployments per day [4]. The testing team needs to devise innovative strategies to expedite the process while minimizing precision loss. This is the meaning of the term "Optimization" in the thesis title and the main challenge of regression testing: maximizing bug detection in the least amount of time.

There are various strategies available to achieve this, which we will now present. The main ones that we will focus on are named *Test case selection* and *Test suite prioritization*. In this thesis, the term *strategy* will be used to denominate the global ways to enhance regression test executions, such as *Test case selection* and *Test case prioritization* presented in 2. On the other hand, the term *technique* will be used to denominate the concrete implementations of these strategies presented in 3 and 4.

1.3 Objectives and Scope of the Study

The objective of this research is to provide a comprehensive overview of regression testing strategies, specifically focusing on techniques for TCP and Test Case Selection (TCS). The study is positioned within the landscape of industry practices, analyzing how regression testing is applied in real-world software development. The focus is primarily set on procedural programming and object-oriented programming and contexts.

Our analysis offers a structured comparison of major methods and tools, highlighting their advantages, limitations, and the trade-offs they entail. The goal is not to present every technique in precise details and our analysis will not focus on recent Machine Learning (ML) techniques, but rather on heuristic, meta-heuristic and algorithmic approaches. Our analysis offers a discussion on evaluating the effectiveness of TCP and TCS strategies. We also discuss the concrete use in today's industry and the remaining challenges.

1.4 Discussing the selection of articles

We selected old and particularly well-regarded articles for the theoretical foundations, while We chose more recent articles to discuss the techniques themselves. Finally, We chose contemporary articles to address current uses and challenges to overcome, or meta-studies to dress a panorama of the current of future *hot* research topics. We tried to support each of my sections with at least one source from the field of research. The articles and papers were generally found on the Google Scholar platform. Whenever possible, We chose articles or papers from high-ranked conferences and journals. The rank is available on the Computing Research Education Association of Australasia (CORE).

Chapter 2

Test Case Selection and Prioritization Strategies: Theoretical Foundations

In this chapter, we discuss the theoretical foundations of the two main regression test suite execution strategies: *Test Case Selection* and *Test Case Prioritization*. We explain why these two are the most prominent and why we chose to focus on them. This discussion also provides an opportunity to distinguish them from other approaches that are less relevant in a regression testing context. Additionally, we present the different methods for evaluating these strategies, along with precise metrics and calculations for each.

2.1 Test Case Selection

2.1.1 Definitions

Test case selection is a well-known problem and many authors have tried to define it. For example, Shi *et al.* [9] and Graves et al. did so [10]. The goal of test case selection is to *reduce* the test suite to only keep the test that are sufficient to detect changes added to a program. For example, in a regression testing context, it would be surprising that running the test of a module would be pertinent if that test has not been modified. As a formal definition, one can present Test case selection as the process of selecting a subset $T' \subseteq T$, where T is the full set of test cases for a program P, to execute on a modified version of the program P'. The goal of test case reduction is to minimize the size of T' while preserving the ability to detect regressions or failures caused by the modifications in P'.

2.1.2 Links with Test Case Reduction

Test case selection and reduction are often confused together, but it's important to clarify the difference between them.

Test Case Reduction Definition

According to Shi *et al.* [9], test reduction aims to remove redundant tests from a test suite, creating a reduced test suite that is a subset of the full test suite. Of course, the objective is to catch the same amount, or almost the same amount, of abnormalities than if we decided to run all the test suite. A formal definition would be the following: Let O denote the full test suite, and let $\text{req}(T)$ represent the set of requirements satisfied by a given test suite $T \subseteq O$. The objective of test-suite reduction is to identify a reduced test suite $R \subseteq O$ such that $\text{req}(R) = \text{req}(O)$. This ensures that the reduced test suite R satisfies the same set of requirements as the original test suite O.

Distinction between Test Case Selection and Test Case Reduction

To echo the work from [6], a test suite cannot be reduced into a smaller one, and meet all the requirements if the test plan is *exclusive*.

Test case selection is sometimes confused with test case reduction. It is important to distinguish both, as Shi *et al.* [9] do. Once again, to echo [5] and [6], test case reduction is to software testing what test case selection is to software regression testing. Test case selection *directly* depends on the last modification of the program. It is one of the input of its equation. While test case reduction only consider the requirements and the test suite. Hence, it is important not to confuse both. Some people tried to apply test suite reduction to software regression testing [11, 12], but it is a diversion from its original purpose, and it's not a main strategy to reduce regression testing execution times. That is why a precise study of test case selection over test case reduction is preferred in this thesis.

2.1.3 Metrics

TCS can be evaluated in different ways, and the studies usually use several ones. We came up with a clarifying nomenclature, and propose the two following concepts: Effectiveness and Efficiency. We will present and distinguish them. Moreover, the throughput time of the algorithm should also be considered.

Effectiveness Metrics

Effectiveness measures how well the selected test cases detect defects relative to the total detectable defects in the system, usually detected by the completed test suite. It is a surety measure on test case selection. It is calculated the following way:

$$\text{Effectiveness} = \frac{\text{Defects detected by the new test suite}}{\text{Defects detected by the whole test suite}}$$

Test case selection techniques that consistently achieve 100% effectiveness scores are called *safe techniques*.

Efficiency Metrics

Efficiency measures the reduction in the number of test cases executed while maintaining sufficient coverage. It is calculated the following way:

$$\text{Effiency} = \frac{\text{Test cases executed after selection}}{\text{Initial total test cases}}$$

Throughput Time

One can think that the throughput time is only measured by time taken by the algorithm to decide which tests to run. However, we can draw a more precise measure based on the analysis offered by Gligoric *et al.*. [13]. They consider 3 different phases: The Analysis Phase, The Execution Phase and the Collection Phase. The Analysis Phase is the stage described earlier. The Execution Phase is the phase where the tests are run: it implies loading all the classes and context necessary to run the selected test. The Collection Phase implies collecting useful data for the next executions. As we will see, this Collection Phase is especially useful in the Dynamic Techniques presented in 3.2. Hence, the throughput time is measured by summing the times of these 3 phases.

2.2 Test Case Prioritization

2.2.1 Definitions

As Rothermel *et al.* explain in their work on the topic [14], TCP is the process of optimizing the execution order of test cases in a test suite T to achieve specific goals, as determined by a objective function f. This function f evaluates the permutations of T by considering various factors, such as:

- Code Coverage: It aims to maximize the parts of the code exercised by the test cases.
- Test Case Diversity: It aims to ensure a variety of scenarios are tested.

- Execution Time and Cost: It aims to minimize the resources required to execute the test cases.
- Test Case Importance: It aims to prioritize critical test cases based on their impact.
- Recency of Execution: Take into account whether a test case has recently been executed.

So formally, the goal is to find a permutation T' of T that belongs to the set of all permutations P_T, such that T' maximizes the function f. We aim to identify $T' \in P_T$ such that for all $T'' \in P_T$, the condition $f(T') \geq f(T'')$ holds.

Test case prioritization can be either used in initial testing or regression testing. In the case of regression testing, that is our point of interest in this thesis, prioritization techniques can use information gathered in previous runs of existing test cases to help prioritize the test cases for subsequent runs [15].

2.2.2 Several Goals

Rothermel *et al.* [15] outline several key objectives for test case prioritization. These include increasing the rate of fault detection to uncover defects earlier during regression testing and improving the coverage of the system under test by meeting code coverage criteria more quickly. Additionally, the approach aims to enhance confidence in the reliability of the system under test at a faster pace. Another objective is to prioritize the detection of high-risk faults early in the testing process. Finally, it seeks to identify defects related to specific code changes promptly during regression testing.

2.2.3 Metrics

Generally, TCP is measured by its efficiency, meaning how quickly the test suite detects defects. Also, it is interesting to have a look to the algorithms throughput time, to measure its practical usage.

Efficiency Metrics

TCP is mainly measured by its efficiency. As presented above, many different kinds of objective functions can exist. The kind of efficiency metrics that should be used is generally directly impacted by the objective function. Some precise metrics do not relate to it.

There exists 4 efficiency metrics categories [1]: APFD, Average Percentage of Faults Detected with Cost (APFDc), APFD derivative, and other metrics.

The Main Metric : APFD

APFD is a metric introduced by Elbaum et al. [15] that measures the rate of fault detection per percentage of test suite execution. "The APFD is calculated by taking the weighted average of the percentage of faults detected during the execution of the test suite. APFD values range from 0 to 100" [16]. It varies from 0 to 100 if we consider it as percentage, otherwise it varies from 0 to 1. More formally, here is its calculation formula:

$$APFD = 1 - \frac{\sum_{i=1}^{m} T_i}{n \cdot m} + \frac{1}{2n}$$

Where:

- n is the total number of test cases.
- m is the total number of faults.
- T_i is the position of the first test case that detects fault i.

From an intuitive point of view, it means measuring the average value on the diagram representing the detection failure during the execution flow. An example of these diagrams is shown in 2.1. APFD is the main metric for test case prioritization, for several reasons. First, it is a meaningful techniques (we measure the rate of fault detection accross time: it seems meaningful if we refer to the test case prioritization definition 2.2.1). Then, it is a general technique (it's hard to imagine a case where this technique does not make sense). Finnaly it is a simple metric to calculate. Hence, this metric is the one that is generally used in the articles, that is quite convenient to compare techniques among them. However, other metrics also exist and are often derivated from this one.

APFDc

However, APFD comes with two defects [16]. Firstly, it assumes that all faults have equal severity. A fault in the payment system is way more problematic than a request with a slightly different HTTP return code. Then, it also assumes that all tests have equal cost. However, some tests might be more resource-demanding : for example integration tests are generally more costly than unit tests.

APFDc aims to resolve these two issues, by giving each test a severity and a cost coefficients. This technique was introduced in the same year as APFD by Elbaum et al. [17]. It can be formally defined as follows [17]: "Let T be a test suite containing n test cases with associated costs $t_1, t_2, \ldots, t_n$. Let F be a set of m faults revealed by T, with severities $f_1, f_2, \ldots, f_m$. Let TF_i represent the first test case in an ordering T' of T that reveals fault i". APFDc is then calculated using the equation:

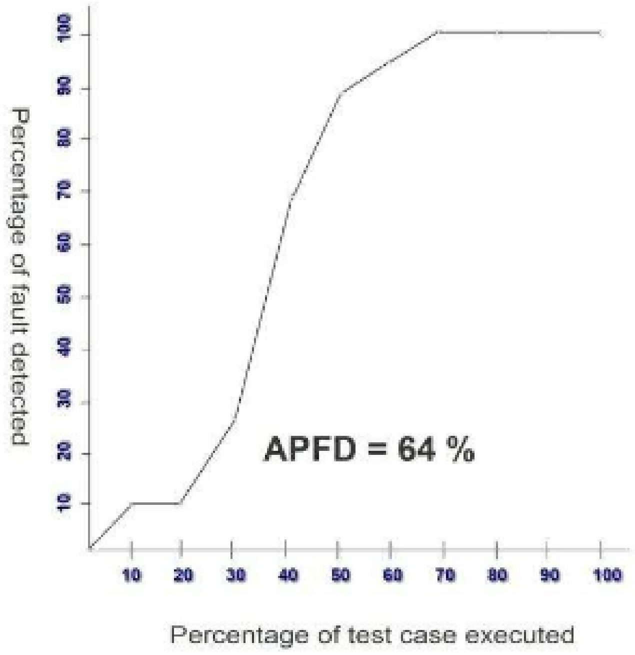


Figure 2.1: APFD diagram example from [1]

$$APFD_C = \frac{\sum_{i=1}^{m} \left(t_i \times \left(\sum_{j=1}^{n} t_{r_{ij}} - \frac{t_i \times t_{r_{ii}}}{2} \right) \right)}{\sum_{i=1}^{m} t_i \times \sum_{i=1}^{m} f_i}$$

Where:

- n: The number of test cases in the test suite T.
- m: The number of faults revealed by T.
- t_i: The cost of executing test case i.
- f_i: The severity of fault i.
- $t_{r_{ij}}$: The cost of the test cases run before fault i is detected.
- T': The prioritized ordering of the test suite T.
- TF_i: The first test case in T' that detects fault i.

If each test have the same cost and severity, the formula becomes the APFD one. This APFDc formula is way more precise and consistent than the APFD one. However, it has two main drawbacks: we need to be able to give each test a precise severity and cost. This operation can easily become time-consuming and costly, and a wrong estimation can lead to bad results

APFD Derivatives

Some efficiency metrics directly come from APFD but are applied on other objective functions. In APFD, we aim to detect faults. But there exists other metrics that mainly interested about coverage. Here is a brief list from [18]:

- Average Percentage of Statement Coverage (APSC): This measures the rate at which a prioritized test suite covers the statements.
- Average Percentage of Branch Coverage (APBC): This measures the rate at which a prioritized test suite covers the branches. This metric is also represented as Average Percentage of Block Coverage.
- Average Percentage of Loop Coverage (APLC): This measures the rate at which a prioritized test suite covers the loops.
- Average Percentage of Condition Coverage (APCC): This measures the rate at which a prioritized test suite covers the conditions.

These metrics are rarely used in regression testing as they are less relevant that APFD to quickly detect faults. They can be used in precise cases where we quickly need to reach a coverage criterion (for example, we may need to reach a 100% criterion just to check if no major exception is launched).

Other Metrics

One can still imagine an infinite amount of other effectiveness metrics, but we have already stated the ones that are almost always used. For example, there also exists the Problem Tracking Report (PTR) metric that is briefly presented in [1] PTR metric that aims to compute when all the faults are finally revealed.

2.2.4 Throughput time

The analysis offered by Gligoric [13], presented in the section 2.1.3, to measure the TCS throughput time is also relevant for TCP. However in TCP, the Execution phase only measures the number of tests to run, as all the classes and context should still be loaded contrary to TCS.

2.3 Complexity Classes of Both Problems

Test case reduction and test suite prioritization are problems that are both NP-complete, as test case selection and test suite prioritization can be reduced to well-known NP-complete problem, and these well-known NP-complete problems can be reduced in our problems (Set-Cover problem [19] for test case reduction and Knapszak problem for Test case prioritization [20]). Hence, one can deduct that as the test suite grows, the time to resolve the problem explodes. However, the main point of applying these strategies is to reduce the regression testing execution time. Then, it looks mandatory to implement interesting and efficient techniques to make the whole process efficient.

2.4 Conclusion

We have presented TCS and TCP: their goals, their formal and informal definitions and distinguished them from related techniques. We have also presented the different metrics to evaluate these techniques: the effectiveness and the efficiency for TCS and the several different metrics to evaluate the TCP strategy: the main one being APFD and the vast majority of other derivate from this one. We have also discussed the whole throughput time of these strategies and the complexity of both problems

Chapter 3

Test Case Selection Techniques: Use cases, Benefits and Drawbacks

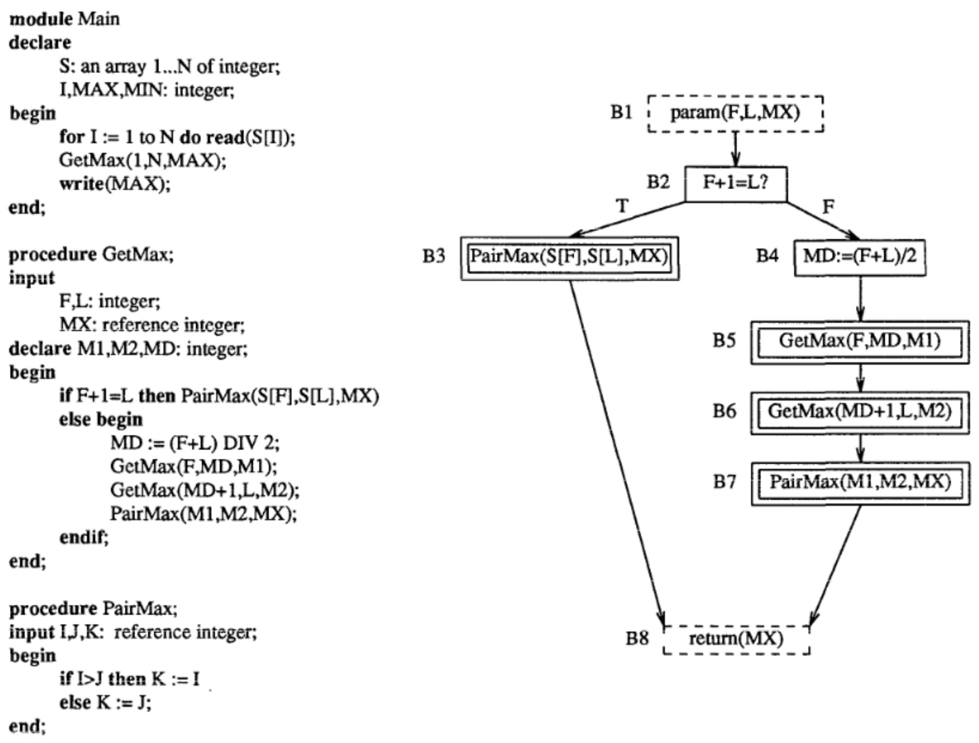
In this chapter, we present concrete techniques that implement the *Test Case Selection* strategy introduced in 2.1. Beyond a simple description of these techniques, this chapter aims to clarify their use cases, advantages, and limitations, as well as to draw comparisons between them to some extent. The techniques are categorized into two main groups: *static* and *dynamic*. They are then categorized by their specificity. Finally, the issue of *safety* is also addressed in this chapter.

3.1 Static Techniques

To quote Anders Moller and Michael I. Schwartzbach [21], static analysis is "the art of reasoning about the behavior of computer programs without actually running them". It means that the first test case selection techniques that we are going to study are based on analysis of the source code of the program and the tests. We see that it studies the program's execution flow, the file dependencies and it can build abstract representations of the program. Yet, it can still collect data to enhance the next executions. However to be considered as a static technique, this data collection should not be mandatory to any execution and should only enhance the process.

3.1.1 Dataflow Techniques

The DFG structure is presented by Orailoglu *et al.* in [22]. An example DFG and its corresponding program can be found in Figure 3.1. A DFG is a directed graph that represents the flow of data through a program. It consists of nodes and edges. The nodes represent program instructions or methods, while the edges represent data dependencies between the edges. Hence, DFG are really useful in static program analysis and is widely used in testing analysis, for example by Harrold *et al.* [23]. In their work on testing analysis, they explain that this structure can easily be used to implement the TCP strategy. Tests are selected only if they execute code portions that is dependent (directly or not) of a code statement that has been modified, added or deleted. For that, and for any other cases of dependency analysis, a dependency graph is built. An example can be found in Figure 3.2 from [2]. The squares are units of code (here these are code statements but other units exist as we will see in next sections), and the circles represents tests. Here, the red squares are in the first layer are the units of code that has been modified, and the red circles the tests that are dependent of these units.



```
module Main
declare
        S: an array 1...N of integer;
        I,MAX,MIN: integer;
begin
        for I := 1 to N do read(S[I]);
        GetMax(1,N,MAX);
        write(MAX);
end;

procedure GetMax;
input
        F,L: integer;
        MX: reference integer;
declare M1,M2,MD: integer;
begin
        if F+1=L then PairMax(S[F],S[L],MX)
        else begin
                MD := (F+L) DIV 2;
                GetMax(F,MD,M1);
                GetMax(MD+1,L,M2);
                PairMax(M1,M2,MX);
        endif;
end;

procedure PairMax;
input I,J,K: reference integer;
begin
        if I>J then K := I
        else K := J;
end;
```

Figure 3.1: DFG Example

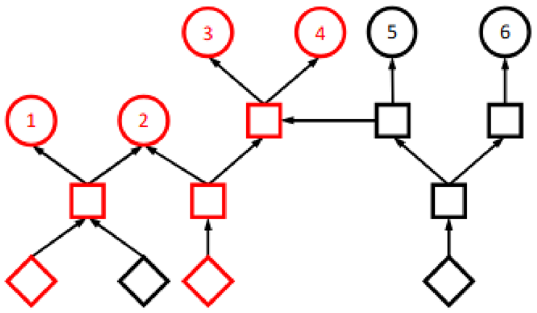


Figure 3.2: Dependency Graph

Graves studies this technique in his test case selection techniques study [10]. However they drew the conclusion that this method has reasonnable results, but is quite costly as it's necessary to draw the entire program dataflow. He explains that this technique is interesting only if the DFG is also used for other things, such as identifying portions of code that are not tested enough. Despite being a simple technique, the global problem lies in the granularity: it is too small, making it a costly technique but and not leading to worth results.

3.1.2 Static Techniques using Files' Dependencies

Researches have tried to tackle the DFG issue using a bigger granularity. A very interesting work using this idea has been proposed by Gligoric *et al.* with his framework Ekstazi [13, 24]. The idea is to increase the granularity going from a method granularity to a file granularity. A dependency graph among files is then drawn. Each tests is assgned to a list of files that are concerned by it. When the TCP stage comes, a list of files that has been modified, deleted of added is available. Then, like in 3.1.1, only the tests that are concerned by a file that is directly or not dependent from a modified file should be run. The modified files are detected computing the checksum of bytecode files.

There are several advantages of this approach. Firstly, the algorithm is more lightweight than the last one. Firstly, the throughput time (cf. 2.1.3) is much lower in this technique. The granularity is rougher, hence the dependency graph is smaller than the DFG. So, the Analysis Phase is way quicker. Then, the Execution Phase is also quicker because for each file not loaded, the class in this file should not be loaded. Finally, the Collection Phase for this technique is quite quick. Only the file graph dependency should be drawn. According to [13], the effectiveness of the technique is not worse than more classical technique, and so could be worth the trade-off. The implementation of this technique led to the creation of the Ekstazi program [13, 24] and can be loaded as a Maven dependency. Its use in industry will be discussed in 5.2.1.

3.1.3 Hybrid approaches: Beyond the Granularity Problem

Many other granularities are possible: a block-level granularity [25] or even a module-level granularity [26], etc. It is not necessary interesting to develop each of them. But an idea emerges from them: the more the granularity is precise, higher the results are but more the technique is heavy and costly. But can we imagine a technique that combines the advantages of the different granularities ? That's the goal of the Hybrid approach proposed by Zhang *et al.* in [27]. The Hybrid Regression Test Selection (HyRTS) approach proposed by Zhang *et al.* aims to balance precision and efficiency by integrating multiple levels of granularity in the test selection process. Instead of relying solely on file-level or method-level dependency tracking, HyRTS combines both to optimize regression testing while controlling computational overhead. We describe algorithm with the three phases we presented earlier with Analysis-Execution-Collection (AEC) in 2.1.3. The analysis phase is performed in two stages: The file-level analysis identifies modified files and selects all test cases covering those files, ensuring broad coverage while maintaining efficiency. The method-level analysis tracks dependencies at a finer granularity, allowing for more precise test selection by identifying the exact methods affected by changes. The more costly operation is only performed on concerned files. The execution and collection phase directly depends on the configuration that is chosen. The algorithm can be run in *offline mode* or *online mode.*

In *offline mode*, the execution phase runs the selected tests without collecting new dependency data. Once execution is completed, the collection phase is triggered separately to update method-level and file-level dependencies for future regression test selection. This separation ensures that test results are available as quickly as possible, making it particularly useful in scenarios where rapid feedback is prioritized. However, since dependency collection is deferred, the overall regression testing time may be higher due to the additional step. In *online mode*, dependency collection occurs simultaneously with test execution. As the selected tests run, the system dynamically records their dependencies, ensuring that the collected information is immediately available for the next round of test selection. This eliminates the need for a separate collection step, reducing the total time required for the entire process. However, because dependency tracking is performed in real time, this mode may introduce additional computational overhead, potentially impacting test execution speed.

This technique is a very interesting trade-off between the last two techniques studied. It is less costly than the the file-based technique while almost as efficient as the method-based or instruction-based ones. However, the algorithm a little bit longer to set up and a little more complex, as the analysis phase is divided in two stages. Like for Ekstazi, a Maven dependency has been deployed to be tested and used easily.

3.1.4 Model-based Techniques

Model-Base Selection (MBS) is a widely used approach for generating test cases from a model that represents the behavior of a system. It has been extensively applied to test generation [28]. MBS relies on system models such as Unified Modeling Language (UML) diagrams, state machines, or control flow diagrams to derive test cases that verify whether the system behaves as expected. Given a well-designed model, test case selection can also be performed using the model itself.

Studies have explored the application of MBS to TCS [29, 30, 31]. One of the main and safer strategies are used is Coverage-Based Selection [29, 30] which selects test cases based on structural coverage criteria.

Coverage-based selection focuses on ensuring that selected test cases achieve a predefined level of coverage over the model. This can include:

- State coverage (ensuring all states in a state machine are tested).
- Transition coverage (ensuring all possible transitions are exercised).
- Path coverage (executing different paths in a control flow graph).

Other implementations of MBS Case Selection/Reduction exist, such as using Similarity Functions [29, 31] between Tests instead of Complete Coverage. However, they are less related to Regression Testing, and more to Classical Testing and Reduction (cf. in 2.1.2 how we distinguished Reduction from Selection), as they aim to remove similar test cases. In a Regression Testing context, with model coverage, we can simply select test cases that are linked to the modified states or transitions. These model-based techniques are particularly relevant for systems with well-defined specifications, such as safety-critical systems, where achieving high coverage is essential [29]. If the program is not well-specified enough, the TCS may not be safe. Moreover, this technique works specifically well with Model-Based Test Generation, as the Generated Tests can directly be linked to their states or Transitions.

3.2 Dynamic Techniques

On the other hand, dynamic analysis is based on an execution of the program. Dynamic applications use cases for testing for example (that are the subjects of our study) can be found there [32]. Here, we will not need of course to execute the program each time we need a selection. The goal will be to keep track of the execution history to be able to analyze it for the next selections. Compared to static analysis, we see there that we do not only study the source code. These techniques are considered *History-aware*.

3.2.1 Dynamic Techniques using Window Algorithms

Elbaum *et al.* introduced window-based algorithms for Regression Test Selection (RTS) in [33]. Their approach determines whether a test should be executed based on two key factors: its recent failure history and its execution frequency.

Let us examine the algorithm in more detail. After each test suite execution, two key pieces of information are recorded: the set of executed tests and their respective outcomes (pass or fail). Two sliding windows are then maintained to capture these aspects: W_f represents the failure window, tracking recent test failures, while W_e represents the execution window, monitoring how frequently a test has been executed. At any given moment, each test has an associated W_f and W_e value.

To determine test selection, predefined thresholds are set for both windows. A test is scheduled for execution if either its W_f or W_e value exceeds the corresponding threshold. The choice of these thresholds directly impacts the trade-off between cost and safety: lower thresholds increase safety but also raise execution costs. According to [33], an effective trade-off is achieved with a failure window W_f of 96 hours and an execution window W_e of 24 hours. The algorithm manages to select only 10% of the total number of tests, but selects between 70% and 80% of failing tests as some failed tests were selected recently, making it a very efficient technique, but with a disputable effectivity.

So this well-suited for Continuous Integration (CI) environments in non-critical systems (as approximately 25% of failing tests are not selected), where efficient testing (see 2.1.3) is crucial for rapid development cycles.

3.2.2 Predictive Test Case Selection

Predictive Test Case Selection is a data-driven test case selection technique presented by Machalica *et al.* in [2]. It is considered a problem classifier, which belongs to the family of ML problems, where the inputs are code changes and test cases, and the output is whether a test is selected. This technique is the only ML technique presented in this thesis because we found it to be effective, lightweight (compared to other ML techniques), and easy to understand for non-Artificial Intelligence (AI) experts. Moreover it relies on features that are also used by other techniques. The whole process is depicted in Figure 3.3.

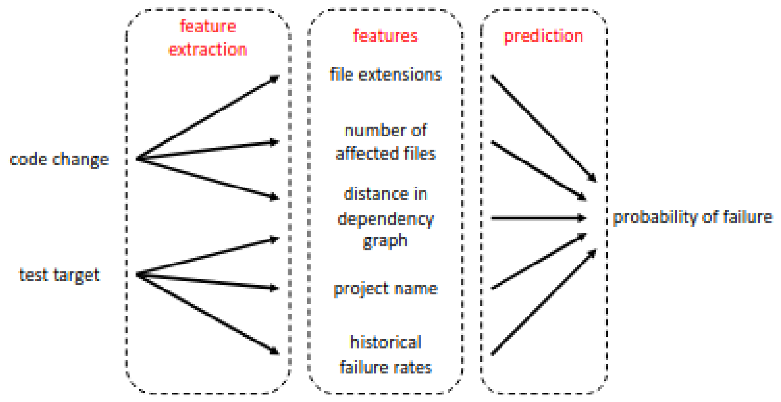


Figure 3.3: Predictive Test Case Selection Process from [2]

Features are extracted from the inputs (including the historical failure rates of the test, which classifies this technique among dynamic techniques), and the probability of failure is computed based on these features. The machine learning model is trained progressively after each test execution, making the technique more efficient with every iteration. As a result, this technique is particularly useful in large-scale projects with long-term development.

The authors demonstrate strong results in both effectiveness and efficiency when the algorithm is well-trained: 99% of faulty code changes are detected, while selecting only one-third of the test cases that a classical build dependency approach would, making it a very powerful technique in Continuous Integration contexts.

However, this approach has some downsides. The first iterations may not be highly efficient or effective, so it may be beneficial to run some initial training iterations. Consequently, this technique might not be well-suited for small projects. Additionally, while the testing infrastructure is less solicited, training the AI model remains costly. Lastly, like other dynamic techniques, this method is not entirely safe. Despite achieving 99% effectiveness, it remains the most effective dynamic technique.

3.3 What technique should you choose?

We have explored various techniques for implementing the RTS strategy, which can be categorized as either static or dynamic. The first consideration is whether test selection must be entirely safe (i.e., 100% effective). If so, a static approach is the preferred choice.

The next factor to consider is granularity. A finer granularity offers greater precision but comes at a higher computational cost, while a coarser approach is more lightweight but may slightly reduce effectiveness. Hybrid techniques provide a middle ground, balancing precision and efficiency. Additionally, model-based techniques are particularly well-suited for well-defined and critical systems where an accurate model is available.

If test selection safety is not a strict requirement—as is often the case in standard projects—dynamic techniques are generally preferable. They tend to be more efficient and are well-suited for modern (CI) environments, where historical data can be leveraged effectively. The choice of dynamic technique depends on the trade-off between effectiveness and efficiency. Window-based algorithms offer strong efficiency and lightweight execution, while predictive test selection, after sufficient training, can achieve near-optimal effectiveness along with competitive efficiency.

Chapter 4

Test Case Prioritization Techniques: Use Cases, Benefits and Drawbacks

In this chapter, we present concrete techniques that implement the *Test Case Prioritization* strategy introduced in 2.2. The plan is the same as the one followed in 3: beyond a simple description of these techniques, this chapter aims to clarify their use cases, advantages, and limitations, as well as to draw comparisons between them to some extent. The techniques are also categorized into two main groups: *static* and *dynamic.*

4.1 Static Techniques

The static techniques holds the same characteristics as the one presented in 3.1. However this time, the techniques used to implement the techniques are different: the TCS static techniques use coverage analysis, requirements and risk analysis (analysis of the program functionalities) and an analysis of the model's program (as in 4 for this one.

4.1.1 Coverage-based Prioritization

Coverage-based prioritization techniques are a classical and fundamental approach presented by Rothermel *et al.* in [14]. The goal is to maximize the coverage as fast as possible, depending on the chosen granularity. Two families of algorithms exist: the greedy ones and the search algorithms.

Greedy Algorithms

Yizhun Wang [34] define greedy algorithms as "algorithms that take the local optimal solution in the current state according to the characteristics of the problems". They are constructive algorithms, that build the solution brick by brick. Their main advantages are simplicity, efficiency and ease of implementation. However, they do not always produce general optimal results. Greedy algorithms follow an heuristics that guide their choice for each iteration. The algorithm supposes that following the heuristics will lead to a satisfying solution.

Rothermel *et al.* [14] presents us several heuristics to follow to solve the test suite ordering problem. Firstly, a coverage granularity must be chosen. Rothermel picked branch coverage and statement coverage. They seem reasonable and the ones that are often studied in coverage-based problems. Then, we must also choose how we compute the coverage. A *total* or *additional* approach must be chosen. The total approach sorts the tests according to the number of unit codes they cover. Additional approaches sorts the tests according to the number of unit codes they cover that have not been covered yet. Hence, in additional approaches, the test suite must be resorted after each iteration. The total approaches have a ($O(n \log n)$) complexity, while the additional apparatuses have a ($O(n^2)$) complexity. Li et. al [35] also introduce K-Optimal algorithm. These are additional algorithms, but the number of unit of codes not covered is updated each K iterations. Hence its complexity is also ($O(n^2)$)

Interesting results came from the analysis of the algorithms. The APFD values were studied for that. Surprisingly, it comes that total algorithms are approximately as efficient as the additional ones, except that their worst cases are better. Considering the way the coverage is computed, Rothermel et al. show that additional and total do lead to similar results, except that total have better wost case (so the additional approach is more expensive and less efficient). Li et al. compared additional and 2-Optimal and concluded that 2-Optimal lead to better results. Although these algorithms are easy to implement and cheaper than others, they are not the most efficient ones. Fault-Exposing Potential (FEP) have also been studied by Rothermel *et al.* in their article and lead to better results (where we take into account the probability that the cover units of code lead to a failing test).

Search Algorithms

As stated earlier, greedy algorithms often leads to optimal solutions, especially for complex problems. We also know that TCP is a NP-hard problem as seen in 2.3 and that Search Algorithms are efficient to solve these problems. Search Algorithms are a family of algorithms that can solves optimization problems using heuristics or meta heuristics. Contrary to greedy algorithms, Search Algorithms are non-constructive algorithms. Li et al. [35] considered two different Search Algorithms: Hill Climbing and Genetic Algorithms (GA), and compared them to Greedy Algorithms seen earlier. In this study, they only focused on coverage issues and not on fault revealing. Hence, they considered APBC, APSC and APCC as efficiency metrics.

Hill Climbing is an heuristic algorithm. It has been chosen in its *steepest ascent* form: at each iteration instead of moving to the first neighbor that gives better results for the coverage computation, the algorithm selects the best neighbor it can find among all the neighbors. GA are often fascinating algorithms and this case is no exception. They are metaheuristic algorithms, meaning that they explore a broad solution space using guided randomization rather than following a strict deterministic path. As general GA algorithms, this is defined by these 6 key elements:

1. Initialization: Several possible test suite orders (between 50 and 100 in the experiment) are initialized. They are considered as chromosomes and each test as a gene.

2. Evaluation: Each ordering is assigned a score that is computed by the chosen efficiency metrics

3. Selection: Based on their score, certain individuals are chosen to reproduce, simulating natural selection.

4. Crossover: Pairs of selected individuals exchange genetic information to create new offspring, which inherit traits from both parents.

5. Mutation: Random changes are introduced in some offspring by modifying certain genes (10% chance that a gene is modified in the study), ensuring genetic diversity and preventing premature convergence to suboptimal solutions.

6. Termination: The algorithm continues for a predefined number of generations (between 100 and 300 in the study)

In their study, Additional Greedy Algorithms beats Total ones. It seems that is contrast from the precedent results. But here, only coverage questions are considered, so it's not surprising. While total algorithms prioritizes tests that coverage

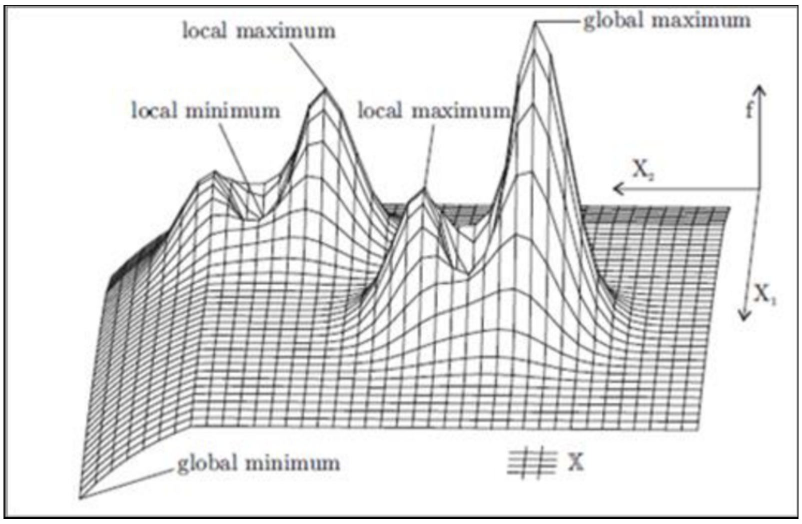


Figure 4.1: X_i are parameters of the test suite ordering, and f is the opposite of the value given by the efficiency metrics.

the most unit of code, hence the ones that may reveal the more faults, additional is only focused on maximizing coverage. The chosen efficiency metrics defines all the results! GA offer competitive results, especially for larger test suites. Hill Climbing, however, struggles with local optima as any other heuristic algorithms, making it less effective as program and test suite sizes grow. To image that, see 4.1. If the starting point of the algorithm is near one of the local optima and if the global optima is further from the local optima than the Hill Climbing scope, it will not be able to detect the global optima and won't move from his local one, despite beeing *steepest ascent*. Metaheuristic algorithms, like GAs, can outperform heuristics in complex scenarios but come with increased computational costs and result variability. Ultimately, heuristic algorithms provide stable and scalable performance, whereas metaheuristics can offer better solutions at the expense of efficiency and predictability.

4.1.2 Requirements and Risk-based Prioritization

Requirement and Risk-Based TCP are closely linked. In both of them, the idea is to prioritize tests depending on the part of the program they are running. These techniques often prioritize depending on high-priority requirements for the first one or depending on areas of the system with higher potential risks for the last one. These techniques are generally particularly useful in critical systems and where compliance or risk management are important, or in projects where the requirements have been entirely well-thought before the development phase. They often take into account the probability of some area of the code to fail, which an idea that we introduced at the end of section 4.1.1.

Requirements-Based

Requirements-Based Prioritization relies on the software's requirements. As Srikanth et al. remind us in their study on requirement and risk-based prioritization in their study [36], the software is directly built upon these requirements, so it looks meaningful to develop TCP techniques also based on them. To illustrate this technique, let us have a look on a classical requirements-based TCP that have been developed by Kovitha *et al.* have developed in their study [3] . Their techniques prioritize the tests according to three different factors:

1. The customer assigned priority of requirements: This measures how important a requirement is to the customer, rated from 1 to 10. Higher priority requirements should be tested early and thoroughly to maximize customer satisfaction.

2. The developer-perceived code implementation complexity: A subjective measure of how difficult a requirement is to implement, rated from 0 to 10 by developers. Higher complexity requirements tend to have more faults, so they require careful testing.

3. The changes in requirements: Tracks how often a requirement has been altered during development, normalized on a scale of 1 to 10. Frequent changes increase the risk of faults, making them crucial for testing.

Weights are applied to each factor, and a null weight can be considered if the factor is not interesting for the software development team. Each requirement is assigned a value from the factors. The article extract detailing the applied algorithm to order the test suite can be found in the following figure Figure 4.2.

Risk-Based

Risk-based TCP can be considered as a Requirements-based TCP derivative. While in the former the focus is set on the requirements properties and the client needs, in the later the focus is more precisely set on the risks linked to the requirements. New factors are introduced such as the Fault Proneness (FP). Srikanth *et al.* [36] define this factor as the one that "provides a measure of how faulty legacy requirements were in a prior release. Fault proneness uses both field failures and testing failures. Ostrand *et al.* have shown that test efficiency can be improved by focusing on the functionalities that are likely to contain higher number of faults [37]".

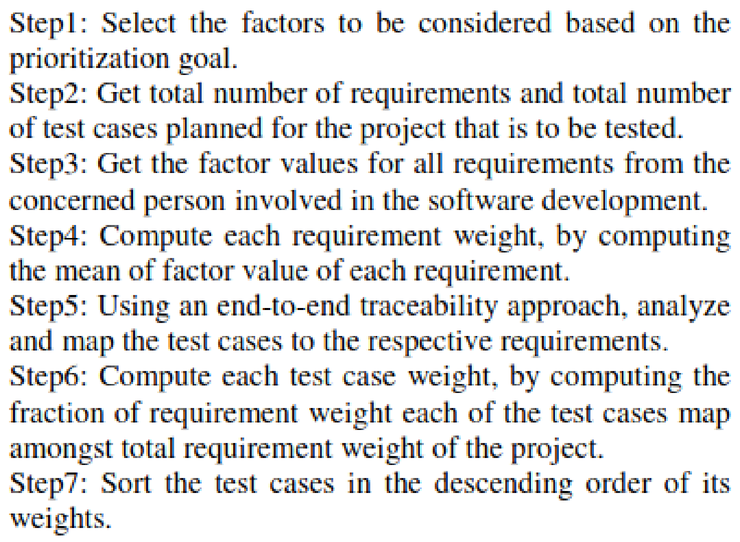

Step1: Select the factors to be considered based on the prioritization goal.
Step2: Get total number of requirements and total number of test cases planned for the project that is to be tested.
Step3: Get the factor values for all requirements from the concerned person involved in the software development.
Step4: Compute each requirement weight, by computing the mean of factor value of each requirement.
Step5: Using an end-to-end traceability approach, analyze and map the test cases to the respective requirements.
Step6: Compute each test case weight, by computing the fraction of requirement weight each of the test cases map amongst total requirement weight of the project.
Step7: Sort the test cases in the descending order of its weights.

Figure 4.2: Requirement-Based TCP from [3]

The values set to the risk factors are submitted to human judgement and may not be precise enough. moreover, it might me time-consuming. Hence, This Hettiarachchi et al. [38] proposed to use a fuzzy expert to automatize the weight values calculations. Contrary to classical logcial systems, fuzzy logics allows reulsts to be represented as probabilities or degrees of truth. Here, the fuzzy expert will compute weights of risk. In their study, they even considered four different risk factors: requirements complexity (RC), requirements size (RS), requirements modification status (RMS), and potential security threats (PST). We can see that in section 4.1.2 the factor "Changes in Requirements" was a first risk factor that can even be used in requirements-based TCP, reinforcing the idea that these two techniques are closely related.

4.1.3 Model-Based Prioritization

Model-based prioritization is related to the model-based selection technique studied in the section 3.1.4: the idea is not to directly work with the code base, but rather with a model or representation of it. Some examples of these models are Simple DirectMedia Layer (SDL) or Extended finite-state machine (ESFM) which consists of states and transitions between states - each transition is defined by: an event, a condition, and a sequence of actions. Model-based Prioritization based on ESFM is studied by Korel et al. in their study [39]. The first and basic Model-Based TCP presentend prioritizes tests that run modified transitions over tests that do not run any.

Beyond this first simple algorithm, the authors present some improved ones. They present Model-Dependence Based Test Prioritization. This algorithm uses dependency analysis as it was done in the section 3.1.2, and aims to identify all the transitions that are worth considering:

1. Affecting Interaction Pattern: Identifies transitions influencing an added/modified/deleted transition via backward traversal, removing untraversed dependence edges.

2. Affected Interaction Pattern: Identifies transitions impacted by an added/-modified/deleted transition via forward traversal, removing untraversed dependence edges.

3. Side-Effect Interaction Pattern: Detects new or removed dependencies introduced by the change.

This algorithm improves upon the first by not only considering modified transitions but also the transitions impacted by the modifications. The more a transition is affected by the changes, the higher the priority given to the related test cases.

The authors also introduce other Model-based algorithms that rely on simple heuristics. The heuristics presented prioritize test cases based on their interaction with modified transitions in a software model.

1. Heuristic 1 prioritizes tests that execute the highest number of modified transitions, assuming that a greater number of executed modifications increases fault detection probability.

2. Heuristic 2 refines this by interleaving selections from different priority levels, allowing lower-priority tests to be chosen earlier.

3. Heuristic 3 shifts focus from the number of modified transitions executed to their execution frequency, favoring tests that trigger modified transitions more often. This heuristic shifts the technique from the static paradigm to the dynamic one.

4. Heuristic 4 modifies this by integrating a more balanced selection across priority levels.

5. Heuristic 5 takes a different approach by ensuring all modified transitions are executed as evenly as possible, prioritizing tests that cover the least-executed transitions first.

These heuristics aim to enhance fault detection efficiency while maintaining a balance in transition coverage. These Model-Based TCP techniques are particularly useful for well-defined systems where a clear model is already available or can be extracted. It is also useful in critical systems, where the critical transitions can be easily identified, and where the endeavor can be focused on.

4.2 Dynamic Techniques : Fault-based Prioritization

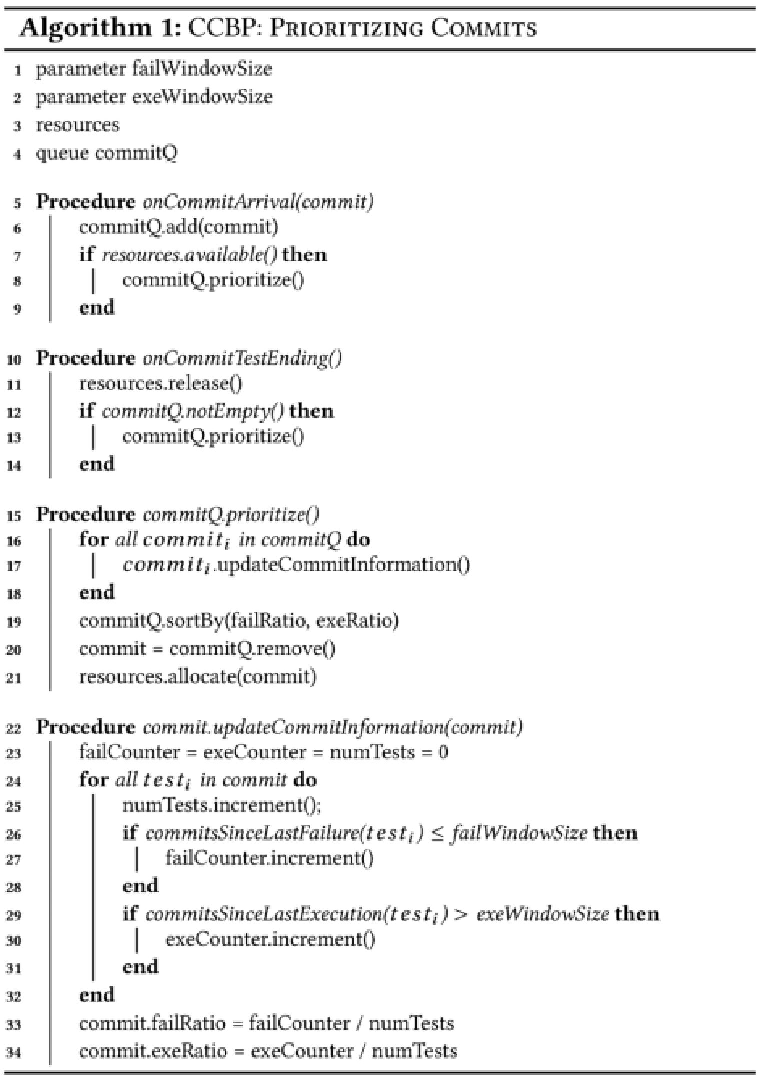
The dynamic techniques for TCP holds the same characteristics as the ones presented for TCS in the section 3.2. However this time, the TCP dynamic techniques mainly use test fault analyses. We name this type of prioritization Fault-Based Prioritization. It orders test cases based on their historical fault detection capability. The idea is to prioritize test cases that have a higher likelihood of detecting faults, especially in areas of the code that have been problematic in the past. It can be easily used in CI and is suited for this environment.

4.2.1 Test-based

Elbaum *et al.* presented how to select test cases based on the execution history in [33] as we have seen in 3.2.1. The new algorithm still uses the execution window (We) and the failure window (Wf) : tests that have not been executed for some time and tests that have failed recently are assigned a high priority. Moreover, the algorithm introduces a new window (Wp): the prioritization window. The window determines when test cases are reassessed for prioritization, either by time or number of test cases. A high value for this window allows for more comprehensive batch processing, reducing the frequency of prioritization and potentially saving computational resources. However, it may delay the execution of critical tests that could identify faults early. On the opposite, a low value for this window ensures that tests are prioritized more frequently, potentially catching issues sooner. Nevertheless, it can lead to more frequent reassessments, increasing overhead and possibly overwhelming the system with constant adjustments. This algorithm is well efficient, but the resource demands scales quite quickly with the number of tests asking to be ordered.

4.2.2 Commit-based

As stated previously, fault-based algorithm are easily used and efficient in modern CI environments. A common and basic granularity in there is the commit, hence new techniques have emerged focusing on this granularity. The goal is not to individually order the tests anymore but rather the commits. Such an algorithm is presented by Liang *et al.* in [4]. They named it CCBP (Continuous Commit Based Prioritization) and has four distinct characteristics: Commit-focused, Fast, Continuous and Resource-aware. The commits arrive and are prioritized continuously in the



Algorithm 1: CCBP: PRIORITIZING COMMITS

```
parameter failWindowSize
parameter exeWindowSize
resources
queue commitQ

Procedure onCommitArrival(commit)
    commitQ.add(commit)
    if resources.available() then
        commitQ.prioritize()
    end

Procedure onCommitTestEnding()
    resources.release()
    if commitQ.notEmpty() then
        commitQ.prioritize()
    end

Procedure commitQ.prioritize()
    for all $commit_i$ in commitQ do
        $commit_i$.updateCommitInformation()
    end
    commitQ.sortBy(failRatio, exeRatio)
    commit = commitQ.remove()
    resources.allocate(commit)

Procedure commit.updateCommitInformation(commit)
    failCounter = exeCounter = numTests = 0
    for all $test_i$ in commit do
        numTests.increment();
        if commitsSinceLastFailure($test_i$) $\leq$ failWindowSize then
            failCounter.increment()
        end
        if commitsSinceLastExecution($test_i$) $>$ exeWindowSize then
            exeCounter.increment()
        end
    end
    commit.failRatio = failCounter / numTests
    commit.exeRatio = exeCounter / numTests
```

Figure 4.3: CCBP algorithm from [4]

algorithm. However, a strong hypothesis is made: the commits are considered independent. It seems that this hypothesis is meaningful for wide projects with several developers or teams working on it (and so on different modules). For other cases, the developers can specify the dependence with other commits, in which cases, the constraint forbidding to prioritize the commits over this one is written. The commits are prioritized in a similar fashion than the tests in the previous algorithm, but with a higher granularity. When a new commit arrives, they are re-organized calculating their Execution Ratio and Fail Ratio. These ratio are calculated from the number of tests exceeding the Execution and Fail Windows ((Wf) and (We)) over the total number of tests in the commit). When resources are available, a new commit is launched. The detailed algorithm is presented in the figure 4.3

To conclude, the CCBP algorithm is a more lightweight one than the previous one seen. The authors observed that the APFDc values for their algorithm are not too different than the classical fault-based prioritization algorithm, concluding that "Prioritizing test suites within commits has limited impact". However, as we have seen a independency hypothesis is needed. It can be overcome but the developer need to inform the dependencies. Moreover, if too many constraints appear, the algorithm becomes less efficient become it is less free to order the suite.

4.3 What technique should you choose?

We have explored various categories of test case prioritization techniques, including Coverage-Based, Risk/Requirements-Based, Model-Based, and Fault-Based approaches.

Model-Based techniques are particularly effective for well-defined systems and critical applications where an accurate model can be extracted. Their effectiveness depends on the correctness of the model itself. Similarly, Requirements and Risk-Based techniques work well in structured environments where requirements are clearly documented, and both clients and developers can accurately identify potential risks. Without well-defined requirements or risk assessments, these techniques lose their reliability.

Coverage-Based techniques are widely applicable across different projects and environments, generally producing reliable results. However, their efficiency varies: Greedy algorithms, while straightforward, may not yield the most optimal prioritization, whereas more advanced approaches like Hill Climbing and GA can achieve better results but may require significant computational effort.

Fault-Based prioritization is also a versatile approach suitable for most projects. Modern CI environments provide robust tools to facilitate its implementation, enabling early fault detection and efficient test execution. However, for projects with very large test suites, these techniques may demand significant infrastructure resources to function effectively.

Chapter 5

Conclusion

5.1 Summary

We have broadly presented and defined TCP and TCS strategies and the ways of evaluating them. TCS is evaluated based on its efficiency, effectiveness, and throughput time, while TCP is evaluated on its efficiency and throughput time only. Then, we have offered an overview of the different techniques (which in our case could be understood as a strategy implementation). The techniques were categorized into static and dynamic approaches. Each of the techniques have its own benefits, drawbacks, and related use cases. Typical benefits or drawbacks include complexity, efficiency/effectiveness, resources needed to run the techniques, the scaling, or the level of abstraction.

In this thesis, we selected an important number of articles that we thought were relevant overviews of the research field on this topic (except the recent articles discussing modern AI/ML techniques). For each of these concepts, we referred to at least one related work. However, we found out that it was difficult to strictly compare techniques with others in numerical terms and be able to affirm that some technique is x% more efficient than another. Indeed, techniques were generally only compared to baseline techniques and not between each other and as the experimental setups were often different.

5.2 Current uses

Now that we have broadly discussed TCS and TCP, it is important to examine their role in today's industry and their potential impact on the future.

5.2.1 Adoption in the industry

Test case selection and prioritization techniques are commonly used by large companies that need to manage vast amounts of testing. These strategies are essential for optimizing time and resources. However, according to Gligoric *et al.* [13], TCS has not yet been widely adopted by standard companies and development teams. In practice, many developers still perform test case selection manually.

In their study on automated vs. manual TCS practices [40], Gligoric *et al.* found that more than half of the developers relied on ad-hoc manual selection. Moreover, manual selections were less efficient than automated ones: on average, developers selected 73% more tests than necessary.

Now, let us explore how these strategies can be effectively implemented in a project.

5.2.2 Implementation in the Continuous Integration

Nowadays, TCS and TCP are used in modern CI environments, where processes can be automated and launched after each action such as commits or merges. The techniques can be implemented with classical CI tools (GitHub Actions, Gitlab CI, Travis) and scripts. The developer should write jobs that are run after each commit that execute some scripts that will be used in selection and prioritization. In the context of History-based techniques, it can write, read and store the results of the tests to use these techniques efficiently. Using these tools allows complete control, flexibility, remove dependencies but needs a complete set-up. On the other hand, the developer can also use external tools and platforms to help them managing their test suite, such as Test Rail[1] which is a platform that aims to help to manage the test suite and to create efficient pipelines.

5.3 Current challenges

To become more efficient and popular among developers TCS and TCP are constantly facing new challenges. We will discuss some of them in this section. We will firstly discuss an efficiency challenge and then some ideas to improve the accessibility issues.

[1]Test Rail - https://www.testrail.com/

5.3.1 Flaky Tests

Many of the algorithms discussed earlier assume that test outcomes are deterministic. However, in practice, some tests—known as flaky tests—produce non-deterministic results. These unreliable tests create significant challenges for both the project and developers: they waste development and CI time, reduce system resilience, and introduce confusion due to inconsistent test results.

According to Gruber *et al.* [41], the primary causes of flaky tests include order dependencies, infrastructure issues, network instability, and APIs returning non-deterministic responses. Their study of over 20,000 open-source Python projects found that nearly 1% of the tests exhibited flakiness.

Flaky test behavior weakens the reliability of dynamic techniques [42], which are generally efficient but lack safety guarantees. Recent research seeks to mitigate this issue [2] by identifying and filtering out flaky failures (test failures caused by flaky tests) before applying selection and prioritization algorithms. This is typically done by re-running the test suite multiple times without modifying the project to detect inconsistencies in test outcomes. The more reruns performed, the more accurately flaky tests can be identified and excluded. However, this approach increases resource consumption, meaning flaky test detection should not become more computationally expensive than the selection and prioritization process itself.

5.3.2 Enhance accessibility

As discussed in Section 5.2.1, TCS and TCP have the potential for broader adoption in the industry. Scientists can take several approaches to encourage their use: developers should be supported and

First, a balance between efficiency and implementation complexity should be found for automated techniques. Scientists can develop tools that are lightweight and easy to integrate into existing workflows. For example, Ekstazi has introduced a simple JUnit framework to facilitate adoption. The lighter and more user-friendly a tool is, the more likely development teams are to adopt it. Otherwise, if the setup process is too complex and time consuming, the benefits of test selection or prioritization may be outweighed by the effort required for implementation.

Additionally, developers should be supported and scientists should aim to increase awareness among them about the impact of test execution time can drive adoption. Google, for instance, promotes this awareness through its TAP system [43, 44], a dedicated group responsible for the continuous integration and testing of most of Google's codebase. TAP identifies and shares insights with developers, such as which types of files are more likely to introduce errors and bugs. Raising awareness on these issues not only helps developers reduce the number of bugs they introduce but also fosters a greater interest in optimizing test execution time, as well as in TCS and TCP techniques.

5.4 Emerging trends

TCS and TCP are two dynamic research fields that are rapidly evolving with new techniques and practices. We will see some of them here.

5.4.1 New Techniques

The use of AI/ML-driven approaches for test case selection and prioritization of test cases is gaining significant traction, as presented in the meta-study written on the subject by Pan *et al.* [45]. For example, the number of significant publications that apply ML techniques in Test Suite Prioritization went from 2 in 2016 to 10 in 2020.

These techniques are often very efficient as they extend fault-based and history-based techniques. However, they sometimes rely on ML models that are not convenient to implement in CI environments, and can be much more resource demanding, than more classical or lightweight techniques.

5.4.2 New Fields of Application

The typical use case for test case selection and prioritization is to accelerate the execution of the regression test suite in a continuous integration and delivery process, ensuring that the program's specifications are properly checked. However, there exist approaches that use the speed gain to perform meta-analysis on the program. One promising example of this is to focus on the energy required to execute the tests, which would allow tracing the changes responsible for the performance issues. In this context, test case selection techniques become particularly relevant, as they could focus solely on the tests related to the most recent changes. This idea is discussed as a future avenue for improvement by Danglot *et al.* for their work [46].

References

[1] M.Tech. (Computer Science & Engineering) University Institute of Engineering & Technology, M.D.U. Rohtak and M. A. Mor, “Evaluate the Effectiveness of Test Suite Prioritization Techniques Using APFD Metric,” *IOSR Journal of Computer Engineering*, vol. 16, no. 4, pp. 47–51, 2014. [Cited on pages ix, 8, 10, and 11]

[2] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, “Predictive Test Selection,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, (Montreal, QC, Canada), pp. 91–100, IEEE, May 2019. [Cited on pages ix, 14, 18, 19, and 33]

[3] R. Kavitha, V. Kavitha, and N. S. Kumar, “Requirement based test case prioritization,” in *2010 International Conference on Communication Control and Computing Technologies*, pp. 826–829, IEEE, 2010. [Cited on pages ix, 25, and 26]

[4] J. Liang, S. Elbaum, and G. Rothermel, “Redefining prioritization: continuous prioritization for continuous integration,” in *Proceedings of the 40th International Conference on Software Engineering*, (Gothenburg Sweden), pp. 688–698, ACM, May 2018. [Cited on pages ix, 3, 28, and 29]

[5] I. Hooda and R. Singh Chhillar, “Software Test Process, Testing Types and Techniques,” *International Journal of Computer Applications*, vol. 111, pp. 10–14, Feb. 2015. [Cited on pages 1, 2, and 6]

[6] H. Leung and L. White, “Insights into regression testing (software testing),” in *Proceedings. Conference on Software Maintenance - 1989*, (Miami, FL, USA), pp. 60–69, IEEE Comput. Soc. Press, 1989. [Cited on pages 2 and 6]

[7] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “Devops,” *IEEE software*, vol. 33, no. 3, pp. 94–100, 2016. [Cited on page 3]

[8] Y. Zhu, E. Shihab, and P. C. Rigby, “Test Re-Prioritization in Continuous Testing Environments,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (Madrid), pp. 69–79, IEEE, Sept. 2018. [Cited on page 3]

[9] A. Shi, T. Yung, A. Gyori, and D. Marinov, “Comparing and combining test-suite reduction and regression test selection,” in *Proceedings of the 2015*

10th Joint Meeting on Foundations of Software Engineering, (Bergamo Italy), pp. 237–247, ACM, Aug. 2015. [Cited on pages 5 and 6]

[10] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, pp. 184–208, Apr. 2001. [Cited on pages 5 and 15]

[11] C.-T. Lin, K.-W. Tang, and G. M. Kapfhammer, "Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests," *Information and Software Technology*, vol. 56, no. 10, pp. 1322–1344, 2014. [Cited on page 6]

[12] U. Sivaji, A. Shraban, V. Varalaxmi, M. Ashok, and L. Laxmi, "Optimizing regression test suite reduction," in *First International Conference on Artificial Intelligence and Cognitive Computing: AICC 2018*, pp. 187–192, Springer, 2019. [Cited on page 6]

[13] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight Test Selection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, (Florence, Italy), pp. 713–716, IEEE, May 2015. [Cited on pages 7, 11, 15, 16, and 32]

[14] G. Rothermel, R. Untch, Chengyun Chu, and M. Harrold, "Test case prioritization: an empirical study," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, (Oxford, UK), pp. 179–188, IEEE, 1999. [Cited on pages 7 and 22]

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001. [Cited on pages 8 and 9]

[16] R. Pradeepa and K. VimalDevi, "Effectiveness of testcase prioritization using apfd metric: Survey," in *International Conference on Research Trends in Computer Technologies (ICRTCT—2013). Proceedings published in International Journal of Computer Applications®(IJCA)*, pp. 0975–8887, 2013. [Cited on page 9]

[17] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pp. 329–338, IEEE, 2001. [Cited on page 9]

[18] A. Mor, “Evaluate the effectiveness of test suite prioritization techniques using apfd metric,” *IOSR Journal of Computer*, vol. 16, no. 4, pp. 47–51, 2014. [Cited on page 11]

[19] N. Alon, B. Awerbuch, and Y. Azar, “The online set cover problem,” in *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pp. 100–105, 2003. [Cited on page 11]

[20] H. M. Salkin and C. A. De Kluyver, “The knapsack problem: a survey,” *Naval Research Logistics Quarterly*, vol. 22, no. 1, pp. 127–144, 1975. [Cited on page 11]

[21] A. Møller and M. I. Schwartzbach, “Static program analysis,” *Notes. Feb*, 2012. [Cited on page 13]

[22] A. Orailoglu and D. Gajski, “Flow Graph Representation,” in *23rd ACM/IEEE Design Automation Conference*, (Las Vegas, Nevada, USA), pp. 503–509, IEEE, 1986. [Cited on page 14]

[23] M. J. Harrold and M. L. Soffa, “Interprocedual data flow testing,” *ACM SIGSOFT software engineering notes*, vol. 14, no. 8, pp. 158–167, 1989. [Cited on page 14]

[24] M. Gligoric, L. Eloussi, and D. Marinov, “Practical regression test selection with dynamic file dependencies,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, (Baltimore MD USA), pp. 211–222, ACM, July 2015. [Cited on pages 15 and 16]

[25] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, “Regression test selection for java software,” *ACM Sigplan Notices*, vol. 36, no. 11, pp. 312–326, 2001. [Cited on page 16]

[26] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka, “Optimizing test placement for module-level regression testing,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 689–699, IEEE, 2017. [Cited on page 16]

[27] L. Zhang, “Hybrid regression test selection,” in *Proceedings of the 40th International Conference on Software Engineering*, (Gothenburg Sweden), pp. 199–209, ACM, May 2018. [Cited on page 16]

[28] M. Shirole and R. Kumar, “Uml behavioral model based test case generation: a survey,” *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, pp. 1–13, 2013. [Cited on page 17]

[29] H. Hemmati and L. Briand, "An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, (San Jose, CA, USA), pp. 141–150, IEEE, Nov. 2010. [Cited on page 17]

[30] W.-T. Tsai, X. Zhou, R. A. Paul, Y. Chen, and X. Bai, "A coverage relationship model for test case selection and ranking for multi-version software," *High Assurance Services Computing*, pp. 285–311, 2009. [Cited on page 17]

[31] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto, "On the use of a similarity function for test case selection in the context of model-based testing," *Software Testing, Verification and Reliability*, vol. 21, pp. 75–100, June 2011. [Cited on page 17]

[32] M. A. Umar and Z. Chen, "A comparative study of dynamic software testing techniques," vol. 12, pp. 4575–4584, 01 2021. [Cited on page 18]

[33] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (Hong Kong China), pp. 235–245, ACM, Nov. 2014. [Cited on pages 18 and 28]

[34] Y. Wang, "Review on greedy algorithm," *Theoretical and Natural Science*, vol. 14, pp. 233–239, 2023. [Cited on page 22]

[35] Z. Li, M. Harman, and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Transactions on Software Engineering*, vol. 33, pp. 225–237, Apr. 2007. [Cited on pages 22 and 23]

[36] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements based test prioritization using risk factors: An industrial study," *Information and Software Technology*, vol. 69, pp. 71–83, Jan. 2016. [Cited on page 25]

[37] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," *ACM SIGSOFT software engineering notes*, vol. 29, no. 4, pp. 86–96, 2004. [Cited on page 25]

[38] C. Hettiarachchi, H. Do, and B. Choi, "Risk-based test case prioritization using a fuzzy expert system," *Information and Software Technology*, vol. 69, pp. 1–15, Jan. 2016. [Cited on page 26]

[39] B. Korel, G. Koutsogiannakis, and L. H. Tahat, "Model-based test prioritization heuristic methods and their evaluation," in *Proceedings of the 3rd international workshop on Advances in model-based testing*, pp. 34–43, 2007. [Cited on page 26]

[40] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, "An empirical evaluation and comparison of manual and automated test selection," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 361–372, 2014. [Cited on page 32]

[41] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 148–158, IEEE, 2021. [Cited on page 33]

[42] E. Fallahzadeh and P. C. Rigby, "The impact of flaky tests on historical test prioritization on chrome," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, (Pittsburgh Pennsylvania), pp. 273–282, ACM, May 2022. [Cited on page 33]

[43] J. Micco, "Tools for continuous integration at google scale," *Google Tech Talk, Google Inc*, 2012. [Cited on page 33]

[44] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, (Buenos Aires), pp. 233–242, IEEE, May 2017. [Cited on page 33]

[45] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, p. 29, Mar. 2022. [Cited on page 34]

[46] B. Danglot, J.-R. Falleri, and R. Rouvoy, "Can we spot energy regressions using developers tests?," *Empirical Software Engineering*, vol. 29, p. 121, Sept. 2024. [Cited on page 34]

Appendix A

Fiche n°1

A.1 Description de l'article

Titre de l'article : Test Case Prioritization: An Empirical Study

Lien de l'article : link

Liste des auteurs : G Rothermel, RH Untch, C Chu, MJ Harrold

Affiliation des auteurs : Oregon State U., Middle Tenn. State U., Oregon State U., Ohio State University

Nom de la conférence / revue : Proceedings of the International Conference on Software Maintenance, Oxford, UK, September, 1999

Classification de la conférence / revue : A

Nombre de citations de l'article (quelle source ?) : 899

A.2 Synthèse de l'article

Issue Testing software is essential for ensuring good software quality. However, as projects grow larger, so do the test suites. When changes occur in the code, running the entire test suite becomes practically impossible. For example, a test suite containing 20,000 lines of code may require seven weeks to execute. Therefore, prioritization techniques are needed to run the test suite more efficiently. These techniques aim to schedule the test suite to quickly detect new errors, cover all code branches, or achieve other objectives. The goal is to find the most effective prioritization technique.

Context Wong et al. proposed prioritizing test cases based on increasing cost per additional coverage, aiming to detect faults earlier in testing. They focus on prioritizing test cases for execution on modified program versions, using safe regression test selection techniques to select a subset. They don't specify how to prioritize remaining test cases after full coverage. Here, the authors extend coverage-based prioritization, exploring various techniques and emphasizing general prioritization over version-specific approaches.

Prioritization technique goals

1. Increase the rate of fault detection
2. Increase the rate of detection of high-risk faults
3. Increase the likelihood of revealing regression errors to specific code changes earlier
4. Increase their coverage of coverable code in the system
5. Increase their confidence in the reliability

In the article, the authors focus on comparing different prioritzation technique rates of fault detection (the first given goal).

Studied techniques Nine different prioritization techniques are used in the article:

1. Unordered: control technique
2. Random: duplicate control technique
3. Optimal: the best test suite according to our goal
4. Branch-total: prioritize test suites according to the total number of branches they cover (complexity: $O(n \log n)$)
5. Branch-addtl: prioritize in order of coverage of branches not yet covered ($O(n^2)$)

6. FEP-total: Fault-exposing-potential (FEP) refers to the ability of a test case to expose a fault. Statement and branch coverage techniques do not take into account the fact that some bugs can be more easily detected than others and also that some test cases have more potential to detect bugs than others. FEP depends on:
    - Whether test cases cover faulty statements or not.
    - Probability that a faulty statement will cause a test case to fail.
7. FEP-addtl: FEP-total but takes into account all other tests already run
8. stmt-total: prioritize in order of coverage of statements ($O(n \log n)$)
9. stmt-addtl: prioritize in order of coverage of statements not yet covered ($O(n^2)$)

Research questions

RQ1 Can test case prioritization improve the rate of fault detection of test suites ?

RQ2 How do the various test case prioritization techniques compare to one another in terms of effects on rate of fault detection ?

Approach In the study, the authors use different programs, and for each of them generate different test cases. Then, they perform test case prioritization with the given different technique and measure how they perform. To compare different measures, an analogy is made between the potency of an antibiotic and how quickly it kills bacteria, likening it to the speed of fault detection in a test suite. This is quantified using the Average Percentage of Faults Detected (APFD), which is calculated as a weighted average of the percentage of faults detected over the life of the test suite.

Experiment implementation The authors use 7 diverse C programs (from aircraft collision avoidance system to lexical analyzers and pattern matching tool). Each program has multiple versions, each containing one fault, and a large pool of possible test cases. The faulty versions were created manually by seeding faults into the program, aiming for realism and using experience with real programs. Fault seeding was performed by ten individuals, each working independently.

The test pools were populated with black-box and white-box test cases, ensuring coverage of program functionalities and code paths. The generation process involved pseudo-random selection of test cases from the test pool, adding them to the test suite based on branch coverage achieved.

For the experiment, tools such as Aristotle program analysis system for test coverage and control-flow graph information were utilized. Prioritization tools implementing various techniques were developed, and mutation scores for Fault-Exposing-Potential (FEP) prioritizations were obtained using the Proteum mutation system.

Results Optimal prioritization shows a significant improvement in fault detection compared to control and random prioritization techniques.

Across the programs, all heuristic techniques show some level of enhancement in fault detection rate. However, the relative effectiveness of each technique varies depending on the program. Notably, additional FEP prioritization outperforms all prioritization techniques based on coverage, suggesting its potential as a highly effective approach. Furthermore, total FEP prioritization performs competitively, particularly when compared to coverage-based techniques. However, the results are still not reaching the optimal results.

Globally, total heuristic approaches outperforms additional heuristic approaches. Nevertheless, worst cases of additional heuristic are less than the worst cases of total approaches.

Discussion The programs used for experiment were relatively small and larger programs may be subject to different cost-benefit tradeoffs. Also, there is only one seeded fault in every subject program, while in practice programs have much more complex error patterns.

Conclusions and openings Many possible works are possible to go beyond the article:

1. Use other programs and types of test suites.
2. Investigate alternative techniques based on sensitivity analysis.
3. Redefine test case prioritization. We can imagine use multiple objective functions (where here it was only the fault detection rate). It is also conceivable to use prioritization techniques that have some knowledge on the program, like knowing where the last modifications occurred, to perform better prioritization.

Appendix B

Fiche nº2

B.1 Description de l'article

Titre de l'article : An Empirical Study of Regression Test Selection Techniques

Lien de l'article : link

Liste des auteurs : Sebastian Elbaum, Gregg Rothermel, John Penix

Affiliation des auteurs : Los Alamos National Laboratory, Georgia Institute of Technology, University of Maryland, Oregon State University

Nom de la conférence / revue : ACM Transactions on Software Engineering and Methodology, 2001

Classification de la conférence / revue : A*

Nombre de citations de l'article (quelle source ?) : 646

B.2 Synthèse de l'article

Issue The main issue revolves around finding a balance between efficiently selecting regression test cases and effectively detecting defects in the software. Developers need to compromise between the time and resources needed for selecting and executing test cases and the necessity of catching defects introduced by software changes.

Definitions Regression testing is a crucial practice where developers verify that changes made to a program (represented as P', a modified version of the original program P) have not introduced new bugs while still maintaining the desired functionality.
The regression testing procedure is considered like the following:

1. Select a subset of test cases T' from an initial test suite (T) to test the modified program P'.
2. Test suite execution problem: Test P' using T' to ensure the correctness of P' with respect to T'.
3. Coverage identification problem: Create additional test cases T'' if necessary to cover new functionalities or changes.
4. Test suite execution problem: Test P' using T'' to ensure the correctness of P' with respect to T''.
5. Test suite maintenance problem: Create T'', a new test suite and test execution profile for P', from T, T', and T''.

This study focuses on the first step, that is the regression test selection problem.

Context Previous studies have investigated the effectiveness and cost-efficiency of regression test selection techniques in comparison to the retest-all approach, and have offered valuable insights into the practical implications of test selection methods. While evidence indicates that these techniques can save time and resources, further empirical research is necessary to comprehensively understand the advantages and limitations of various approaches. This underscores the need for empirical studies to evaluate the effectiveness of regression test selection techniques in real-world software development scenarios.

Author Intentions The authors intend to conduct an experimental study to assess how different regression test selection algorithms impact the balance between fault detection and test execution costs. They plan to review relevant literature, describe the test selection techniques they will examine, and then design and analyze a series of experiments to compare these techniques.

Studied techniques Different techniques have been proposed to select appropriate test cases for regression testing. These can be broadly categorized as:

1. Minimization technique: Attempts to select minimal sets of test cases from T that cover modified or affected portions of P.
2. Dataflow technique: Selects test cases that exercise data interactions affected by modifications.
3. Safe Techniques: When specific safety conditions are met, these techniques ensure that the selected subset, T', includes all test cases in the original test suite T that can reveal faults in P'. Minimization and dataflow techniques are not safe.
4. Ad Hoc/Random Techniques: Selection based on intuition or randomness.
5. Retest-All Technique: Simply reuses all existing test cases.

Research Questions

RQ1 How do techniques differ in terms of their ability to reduce regression testing costs?

RQ2 How do techniques differ in terms of their ability to detect faults?

RQ3 What trade-offs exist between test suite size reduction and fault detection ability?

RQ4 When is one technique more cost-effective than another?

RQ5 How do factors such as program design, location and type of modifications, and test suite design affect the efficiency and effectiveness of test selection techniques?

The Experiment The authors want to assess the costs and benefits of various regression test selection techniques in software development. They constructed two models: one to calculate the cost of using a regression test selection technique and another to measure the fault detection effectiveness of the resulting test suite. In the cost model, they consider the expenses associated with both test selection analysis and the execution/validation of selected test cases. They compare this cost to that of the retest-all approach, where all test cases are executed and validated. Simplifying assumptions are made, such as uniform test case costs and equivalent units for all subcosts. The fault detection effectiveness model evaluates how well regression test selection techniques maintain the ability to detect faults compared to the full test suite. Two methods are considered:

1. A per-test-case basis, identifying discarded fault-revealing test cases.
2. A per-test-suite basis, categorizing outcomes where fault detection may be compromised.

Studied variables Three independent variables are studied during the experiment:

1. The subject program P
2. The test selection T
3. The criteria used to create the test suite M

With P, P', T, and M, the following measures are taken:

1. The ratio of the number of test cases in the selected test suite T' to the number of test cases in the original test suite T
2. Whether one or more test cases in T' reveals at least one fault in P'

For each combination, 100 associated tests are used. From these 100 data points, two dependent variables are computed:

1. Average reduction in test suite size
2. Fault detection effectiveness

Experiment implementation The authors use nine different programs for their experiment, sourced from Siemens program, Space program (an interpreter for an array definition language within a large aerospace application) and Player program (the largest subsystem of the Internet Game Empire).

The test suites associated are randomly generated or generated using an edge-coverage-adequate technique (using control-flow graph information and test coverage data).

The regression test selection techniques used in the experiment were implemented or simulated to assess their effectiveness in reducing the cost of test execution while maintaining fault detection capabilities. Here's how each technique was applied:

1. Minimization Technique: selects a minimal test suite achieving edge-coverage adequacy for modified code. Tests are selected using the DejaVu tool or manually.
2. Dataflow Technique: involves manually inspecting program modifications to generate a list of affected definition-use pairs. Test cases satisfying these pairs are selected using a dataflow testing tool.

3. Safe Technique: constructs control-flow graph representations of program versions and selects test cases based on whether their execution traces contain "dangerous edges" associated with modifications.
4. Random Technique: selects a given percentage of test cases, providing a baseline comparison for other techniques.
5. Retest-All Technique: no implementation needed.

The experiment uses a full factorial design with 100 repeated measures. For each subject program, 100 coverage-based and 100 random test suites from the test-suite universe are selected.

Results

1. The study shows that minimization consistently resulted in the smallest test suites, often containing only one test case. However, its fault detection effectiveness was relatively low. On the other hand, safe and dataflow techniques demonstrated similar behavior in reducing test suite size. The safe technique performed best on large programs, whereas dataflow performed better on smaller programs.
2. The minimization technique had the lowest overall effectiveness, while random techniques showed improved effectiveness with larger test suite sizes. Safe and dataflow techniques exhibited similar median performances, but dataflow had some outliers where faults went undetected.
3. The mimization technique produced the smallest yet least effective test suites. The safe technique consistently achieved 100% fault detection effectiveness, but the sizes of its test suites varied widely. In situations where analysis costs were lower than the costs of running the unselected test cases, the safe technique proved preferable to running all test cases in the suite.

Appendix C

Fiche nº3

C.1 Description de l'article

Titre de l'article : Search Algorithms for Regression Test Case Prioritization

Lien de l'article : link

Liste des auteurs : Zheng Li, Mark Harman, and Robert M. Hierons

Affiliation des auteurs : King's College London, Brunel University

Nom de la conférence / revue : IEEE Transactions on software engineering, 2007

Classification de la conférence / revue : Q1

Nombre de citations de l'article (quelle source ?) : 1000

C.2 Synthèse de l'article

Issue Greedy algorithms are widely used to prioritize test cases. They use objective functions and heuristics to try to detect errors as early as possible. For example, they focus on maximizing the number of branches, statements, or blocks that are covered. However, we also know that the test case prioritization problem is a NP-hard problem, equivalent to the knapsack problem. Moreover, we know that greedy algorithms may sometimes compute suboptimal solutions Hence, it looks important to also study search algorithms that are often useful in such problems, and to compare them with classical greedy approaches. These algorithms could be heuristic or metaheuristic algorithms, for example genetic algorithms, that have not yet been studied in the prioritization test case problem.

Author intentions The authors want to evaluate the performances of these different kinds of algorithms. They study three heuristic algorithms:

1. Greedy algorithm: find the "next best" according to a cost function. It refers to the classical branch or statement coverage for example.
2. Additional greedy algorithm: a variation of the first one that takes into account the first picks. Considering the branch coverage, it will prioritize tests that explore branches that has not been covered yet (when the first one chose tests that cover the biggets number of branches).
3. 2-Optimal algorithm: It's a variant of the additional greedy variation. Considering a branch coverage, it selects the 2 first tests that cover the biggest number of branches that has not been covered yet. It refreshes the set of covered branches every two tests selected, when the classical additional greedy does it for every test.

They also study two metaheuristic algorithms:

1. Hill climbing: It starts with a random initial solution. It evaluates all neighboring solutions obtained by swapping the first test case with any other test case, moving to the neighbor with the highest fitness improvement. It repeats until no better neighbor is found, returning the current solution as the optimal state.
2. Genetic algorithms: Test case sequences are encoded as ordered arrays, and parameters are tuned based on program size, with larger populations and more generations for larger programs. Crossover and mutation operators are used to ensure diversity and avoid premature convergence.

Research questions

RQ1 Which algorithm is most effective in solving the test case prioritization problem for regression testing?

RQ2 What factors could affect the efficiency of algorithms for the test case prioritization problem for regression testing?

Approach The authors set up an experiment where three parameters are considered:

1. The program to which regression testing is applied.
2. The coverage criterion to be optimized.
3. The size of the test suite.

Then, they use effectiveness measures to evaluate the performances of the different parameters combinations. Different statistical analysis are finally used.

Approach implementation The study parameters are implemented the following ways:

1. The study uses six C programs divided into "small" and "large" categories. Small programs originated from Siemens Corporate Research, while large programs were developed for the European Space Agency and Unix, respectively.
2. Test suites are generated by randomly selecting test cases from a pool to achieve full branch coverage, ensuring both small and large test suites met this criterion.The three coverage criterion that have been chosen are block, decision, and statement coverage.
3. Considering the test suites, two granularities of test suite size were used: 1000 small suites of size 8-155 and 1000 large suites of size 228-4,350. For each program, to reduce the computation time, half of the tests were used.

The different fitness metrics that are used are all based upon APFD (Average of the Percentage of Faults Detected), which measures the weighted average of the percentage of faults detected over the life of the suite. Each criterion is linked to its own fitness metric: APBC (Average Percentage Block Coverage), APDC (Average Percentage Decision Coverage) and APSC (Average Percentage Statement Coverage).

Then, the authors use an analysis tool named Cantata++, to exploit the results and to determine statistical significance. The results are obtained using two different statistical analysis: ANOVA Analysis, and Multiple Comparisons (Least Significant Difference).

Results The authors analyze the results in two different sections. First, they consider the results obtained running the small test suites, and then the results obtained running the large test suites. Moreover, as the size of programs grows, the different test case prioritization results become important. Globally, the classical greedy algorithm is the less effective, and the genetic algorithm is the most effective.

First, the authors analyze the results with small test suites. For the small programs, Genetic algorithms and Additional greedy algorithms performs the best. For larger programs, the best results are hold by additional greedy and 2-optimal algorithms. Metaheurstic algorithms are not efficient for these kinds of parameters. For example, Hill Climbing often falls in local optima. Hence, metaheuristic algorithms looks better when the test suite is proportionally bigger size than the program size.

Then, the authors analyze the results with the large test suites. Once again, there is no significant difference between the best algorithms Additional greedy and Genetic algorithm. For bigger programs, the best results were obtained by Additional Greedy and 2-Optimal greedy. The results are more constant for heuristic algorithms, whereas metaheuristic algorithms results are more variable, but can offer better results. Similarly, the variance is more important when running larger test suites. The deterministic results are scalable to larger programs, as the fitness metrics are computed only once. For the larger programs, cost-benefit tradeoffs must be considered for metaheuristic algorithms.

Overall, heuristic algorithms are efficient and scalable. On the other hand, metaheuristic algorithms may provide better solutions, but with increased computational cost and variability.

Implications from the results Hill Climbing often found local optima results. It happens even more as the size of the program and the test suite grow. This phenomenon implies that the fitness landscape is multimodal. It means that the metaheuristic algorithms can outperform the heuristic algorithms, especially as the size of programs and test suites grow.

Additional greedy algorithm is better than the classical greedy algorithm, as it prioritizes test cases that cover new or uncovered aspects of the program. One can imagine that the 2-Optimal greedy algorithm may outperforms both of them, because it appears to address their weakness. However, even if indeed, the algorithm often reach total coverage earlier than the additional greedy one, it has lower coverage earlier in the process. It emphasises one phenomenon in the efficiency of such algorithm: the importance of "crossover points". It's the point where two APFD lines cross, and can determine the effectiveness of an algorithm over another.

Conclusion The greedy algorithm performs notably worse than the additional greedy, 2-optimal, and genetic algorithms. The 2-optimal algorithm addresses weaknesses of the greedy and additional greedy algorithms, but its effectiveness is not significantly different.

The choice of coverage criterion does not significantly affect the efficiency in test case prioritization. However, the sizes of the test suite and the program influences the complexity of the problem and affect the difficulty to compute fitness values. The problem offers a multimodal fitness landscape, that should benfit to metaheuristic problems. These kinds of algorithms are the one that can offer the best theoritical effciencies. However, the heuristic algorithms more constant and are sometimes more pratically efficient.

Appendix D

Fiche nº4

D.1 Description de l'article

Titre de l'article : Techniques for improving regression testing in continuous integration development environments.

Lien de l'article : link

Liste des auteurs : Sebastian Elbaum, Gregg Rothermel, John Penix

Affiliation des auteurs : University of Nebraska, Google

Nom de la conférence / revue : ACM International Conference on the Foundations of Software Engineering, 2014

Classification de la conférence / revue : A*

Nombre de citations de l'article (quelle source ?) : 424

D.2 Synthèse de l'article

Issue In continuous integration development environments, software engineers frequently integrate new or changed code with the mainline codebase to reduce rework and speed up development. The need for timely testing is critical: to provide developers with quick feedback, and to avoid bottlenecks in the CI, which is a resource-constraint environment. Traditional regression testing techniques, relying on code instrumentation and complete test sets, are often unsuitable for CI due to frequent testing requests and high code churn. The goal is to develop effective regression testing strategies that provide thorough and timely testing in continuous integration environments.

Definitions

- Regression Test Selection (RTS): RTS techniques aim to select a subset T' from T that includes test cases important to re-run. When specific conditions are met, RTS can be "safe," meaning it does not omit test cases that would reveal faults due to modifications in P'.
- Test Case Prioritization (TCP): TCP techniques reorder test cases in T to meet testing objectives, such as revealing faults more quickly. TCP does not discard test cases, avoiding the drawbacks of RTS when safety cannot be ensured. TCP can also work with RTS to prioritize test cases in the selected subset T', ensuring beneficial testing if activities are unexpectedly terminated.
- Pre-submit testing: the phase of testing where developers specify modules to be tested, and where regression test selection techniques are used to select a subset of the test suites for those modules that render that phase more cost-effective.
- Post-submit testing: the phase of testing where dependent modules are tested using test case prioritization techniques to ensure that failures are reported more quickly.

Algorithms proposed by authors In continuous integration environment, a new data is available: the history of past executions. The authors imagine algorithms that use this data to offer efficient algorithms to select and prioritize test suites. They propose to decompose the regression testing process in two phases : the pre-submit and the post-submit phases. Each of these phases is identified by a given algorithm.

The first algorithm proposed is named SelectPRETests, that aims to make pre-submit testing more efficient. The algorithm selects a subset of test suites from a given set based on three criteria:

1. Recent Failures: Test suites that have failed within a specified "failure window" (W_f) are prioritized because they are more likely to reveal issues in the modified code.
2. Execution Frequency: Test suites that have not been executed within a specified "execution window" (W_e) are included to ensure they are not neglected.
3. New Test Suites: Any new test suites are included to ensure that new functionality or changes are adequately tested.

This approach aims to reduce the number of test suites run during pre-submit testing while maintaining high effectiveness, ensuring that only the most relevant tests are executed.

The second algorithm, PrioritizePOSTTests, enhances post-submit testing efficiency by prioritizing test suites based on their failure and execution history. It introduces a prioritization window (Wp) to determine when to prioritize pending test suites in the dispatch queue. Test suites are assigned higher priority if they have recently failed (Wf), not been executed within a certain window (We), or are new.

The algorithms seek to mitigate the bottlenecks and inefficiencies commonly encountered in CI environments. Specifically, the proposed approaches help:

- Provide faster feedback to developers about potential issues.
- Reduce the number of undetected faults.
- Manage the high volume and rapid pace of code changes in large-scale software projects.

Research questions

RQ1 How cost-effective is the RTS technique during pre-submit testing, and how does its cost-effectiveness vary with different settings of W_f and W_e?

RQ2 How cost-effective is the TCP technique during post-submit testing, and how does its cost-effectiveness vary with different settings of W_p?

Experiment Both RTS and TCP components are studied independently during the study. Two independent variables are studied in the experiment : the technique and the windows that are used.

For RQ1, three techniques are used :

1. The technique presented above, which selects a subset of the test suites in the change list for a module M. Three different execution windows We sizes are used (1, 24, 48), and nine failure window sizes Wf (0.25, 0.5, 1, 2, 4, 12, 24, 48, 96), each representing a number of hours.
2. A baseline approach that runs all test suites in the module.
3. A random RTS approach

For RQ2, two techniques are used :

1. The technique presented above. Wf and We are arbitrarly fixed at their median values (12 and 24), and seven value are used for Wp (0.1, 0.5, 1, 2, 4, 8, 12), representing number of hours.
2. A baseline approach that does not prioritize.

For RQ1, three dependent variables are mesured:

1. Percentage of test suites selected: the proportion of test suites chosen by the proposed technique compared to the baseline technique.
2. Percentage of execution time required: the amount of time needed to execute the selected test suites relative to the baseline.
3. Percentage of failures detected: the proportion of detected failures by the proposed technique compared to the baseline. These variables are evaluated for each combination of failure and execution window sizes.

For RQ2, one dependent variable is mesured:

1. Time to failure detection: the duration it takes for test suites to detect a failure.

Experiment implementation The algorithms were implemented in Python, consisting in about 300 lines of code. The study utilizes the The Google Shared Dataset of Test Suite Results (GSDTSR) dataset to simulate a continuous testing environment by processing each line of data as if the test suites were executed in real-time, using the recorded duration and results.

Results For RQ1, as the failure window (Wf) increases, there's a noticeable reduction in the percentage of test suites selected for execution. This reduction is attributed to the algorithm's ability to skip non-failing test suites within the given window. Despite skipping some tests, the technique demonstrates significant gains in detecting failing test suites, especially when Wf is set to 96 hours. Additionally, larger values of the execution window (We) lead to more aggressive test suite selection. This is because larger We values allow for the inclusion of more test suites within the execution window, thereby influencing the selection process.

For RQ2, pioritization techniques outperform the no-prioritization baseline across different window sizes (Wp). Increasing Wp results in a greater number of test suite executions marked as high priority due to the inclusion of more stale data. However, the study shows that prioritization techniques with better median performance also exhibit greater variation. This suggests a trade-off between median performance and variability. Techniques with smaller prioritization windows show better median performance but higher variability, while larger windows exhibit more stable but slightly lower median gains.

Appendix E

Fiche de synthèse

E.1 Analysis Grid

E.2 Development

The regression testing process can be divided into two parts. First, a selection of the tests to run is made, and then, the selected tests are prioritized. There are many different algorithms available, depending on the environment. Classical heuristic algorithms exist for prioritization, where the objective criterion can be tuned. Even metaheuristic algorithms, which are very promising, are used for prioritizing. For selection, different classical algorithms exist, such as minimization, dataflow, or safe techniques. New kinds of algorithms emerge when a history of past executions is available, which is for example the case in continuous integration developments.

The variables used for experiments depend on the environment:

- In a classical environment, they include a program and a test suite. An additional element can be added, such as a specific fault in a program or the kind of objective criterion.
- In a history-based environment, using a history of past executions and results is very useful. First, it allows for the use of historical data, and moreover, any program or test suite generation is not needed anymore, as they are already included.

	Regression testing part	**Studied algorithms**	**Independent variables**	**Studied dependent variables**	**Controle techniques ?**
#1 *Test Case Prioritization: An Empirical Study.*	Prioritization	Classical algorithms	A program, a fault seeded in the program, a test suite	Average Percentage of Faults Detected (APFD)	Yes: Unordered and Random
#2 *An Empirical Study of Regression Test Selection Techniques.*	Selection	Classical algorithms	A program, a test selection, and a criteria used to create the test suite	Average reduction in test suite size - Fault detection effectiveness	Yes: Retest-All technique
#3 *Search Algorithms for Regression Test Case Prioritization.*	Prioritization	Heuristic and Meta-heuristic algorithms	A program, a test suite, and a coverage criteria to optimize	Average Percentage of Faults Detected (APFD) and its derivatives	No: Studied algorithms are only compared between them
#4 *Techniques for improving regression testing in continuous integration development environments.*	Selection and Prioritization	History-based algorithms	Google Shared Dataset of Test Suite Results	Percentage of test suites selected - Percentage of execution time required - Percentage of failures detected - Time to failure detection	Yes: Random test suite selection and No prioritization algorithms

Table E.1: Analysis grid of the studies.

The dependent variables are greatly influenced by the part of regression testing that is studied. The main metric for prioritization is the Average Percentage of Faults Detected (APFD), while studies on selection focus on the effect on the test suite size and the trade-offs between the detection we might lose and better execution time.