

Università degli Studi di Napoli Federico II
Ph.D. Program in
Information Technology and Electrical Engineering
XXXVI Cycle

Thesis for the Degree of Doctor of Philosophy

Churn-Based Approaches for Regression Test Prioritization

by

Francesco Altiero

Advisor: Prof. Adriano Peron

Co-advisor: Prof. Anna Corazza

Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

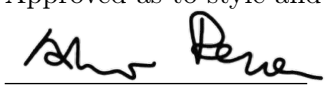
Churn-Based Approaches for Regression Test Prioritization

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

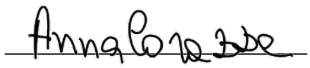
Francesco Altiero

March 2024

Approved as to style and content by



Prof. Adriano Peron, Advisor



Prof. Anna Corazza, Co-advisor

Università degli Studi di Napoli Federico II

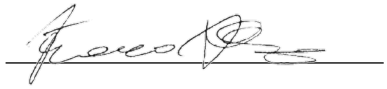
Ph.D. Program in Information Technology and Electrical Engineering
XXXVI cycle - Chairman: Prof. Stefano Russo

http://itee.dieti.unina.it

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
Parts of this dissertation have been published in international journals and/or conference articles (see list of the author's publications at the end of the thesis).

Napoli, March 11, 2024



Francesco Altiero

Abstract

Regression Test Prioritization is a consolidated industrial practice in software evolution scenarios when there are limited resources allocated to the testing phase. It aims to reduce the efforts of Regression Testing by re-arranging the order of execution of test cases to increase the rate at which faults are detected. Various prioritization techniques employ different criteria to permute the test cases, as *test coverage* or meta-heuristics. Few studies in literature consider the code churn, i.e. the modification in the software source code, to drive the construction of the permutation.

To fill the gap, this thesis presents two novel prioritization techniques leveraging on change information to produce a meaningful permutation. The first technique, namely Genetic-Diff, is an approach based on genetic algorithms to search for a permutation which prioritizes test cases covering changed parts of the source code. The second technique quantifies the structural similarity of changed code by means of Tree Kernel functions, giving higher priority to test cases covering more structural changes.

The proposed techniques have been empirically evaluated on benchmark software projects, automatically injected with faults due code mutations. For both techniques, the experiments showed a significant increase in fault-detection performance, compared to several well-known state-of-the-art prioritization approaches.

Moreover, this thesis presents *ReCover*, a novel dataset of software projects with real faults, collected through mining online code repository, to support Regression Test Prioritization researches. Several prioritization strategies have been executed on *ReCover* and their fault-detection performances analyzed and discussed.

Keywords: Regression Test Prioritization, Software Testing, Code Churn, Tree Kernels, Genetic Algorithms, Benchmark Prioritization Dataset

Sintesi in lingua italiana

La Prioritizzazione dei Casi di Test è una pratica consolidata in contesti di evoluzione software, quando le risorse allocate per la fase di test sono limitate. Lo scopo è ridurre il costo dei Test di Regressione mediante la modifica dell'ordine di esecuzione dei test, per aumentare la rapidità di scoperta dei fault. Differenti tecniche di prioritizzazione usano vari criteri per permutare i casi di test, come *test coverage* o meta-euristiche. Pochi studi in letteratura considerano le modifiche al codice sorgente (code churn) del software per costruire le permutazioni.

Per colmare il vuoto, questa tesi presenta due nuove tecniche di prioritizzazione che impiegano le informazioni sui cambiamenti per produrre una permutazione utile. La prima, Genetic-Diff, è una tecnica di tipo genetico che cerca una permutazione in cui i test che coprono parti modificate nel codice abbiano alta priorità. La seconda fa uso di Tree Kernel per quantificare i cambiamenti strutturali all'interno del codice, al fine di dare maggiore priorità ai test che coprono la maggiore entità di cambiamenti strutturali.

Le tecniche proposte sono state analizzate empiricamente su progetti software di benchmark, a cui sono stati iniettati fault attraverso mutazioni del codice. Per entrambe le tecniche, gli esperimenti hanno mostrato un significativo incremento nella velocità di scoperta dei fault, comparati a varie strategie di prioritizzazione allo stato dell'arte.

La tesi descrive inoltre *ReCover*, un nuovo dataset di progetti software con fault reali, estratti da repository di codice pubbliche, per supportare la ricerca in Prioritizzazione dei Casi di Test. Varie tecniche di prioritizzazione sono state sperimentate sul dataset, e le loro prestazioni discusse.

Parole chiave: Prioritizzazione dei Casi di Test, Test di Software, Code churn, Tree Kernels, Algoritmi Genetici, Dataset per Benchmark di Prioritizzazione

Acknowledgements

Summarizing three intense years in few pages is not an easy task. For me, every single word in this thesis carries all the experiences and the struggles and the failures and the successes which I encountered during this wonderful journey. Thus I will try to transmit at least a part of what I experienced through these few words of this acknowledgements section. And, of course, to thank all the people who went along with me in this adventure.

Among all, I want to sincerely thank my PhD supervisor, Prof. Adriano Peron, and my co-supervisor, Prof. Anna Corazza, who followed me closely during all this experience. From Prof. Peron I learned how much exciting doing research could be, and he transmitted me the interest in drilling down each idea and result we obtained to reason thoroughly about everything which was unclear. From Prof. Corazza I learned how to be critical and to not stop on the surface, due to her precious advice which led me to think twice and be more precise and concise (and, also, to made me realize the fact that this thesis was not going to write by itself...).

I want also to thank Prof. Sergio Di Martino, whose hints taught me how to be more pragmatic both in the definition of goals and in communicating my ideas, and Dr. Luigi Libero Lucio Starace, who has been an invaluable support in discussing ideas. My gratitude goes also to Prof. Stefano Russo, chairman of my PhD course, whose consideration and attention to us PhD students was unmatched, and Adriana D'Auria, who had been always available and kind in directing me through the inevitable bureaucracy and paperwork.

Furthermore, I want to thank everyone in the Software Engineering Group of the Technische Universitä Wein, and in particular Prof. Jürgen Cito, for hosting me during my period abroad and for making me feel at home. And I want also to thank Prof. Breno Miranda and Prof. Leonardo

Mariani for their precious reviews to this work, which allowed me to revise some unclear points and to improve its overall quality.

Besides all who directly sustained my in the works of these three years, there are a lot of people who helped me indirectly, and were helpful for my sanity. The first of these has been Vincenzo, who is simply the best friend anybody could ever wish to have. Then, there is Claudio, who is the other best friend everyone should have. Then, I want to thank Sara, Renato[1], Valentina, Giovanni, Federica, Louise, Giuliano, Mario, Simone and a lot of other people (excluding Ilia) whose listing may cause this acknowledgement section to grow larger than the actual thesis itself. I want also to thank my father, my mother, my sister and my brother to supporting my choices and helping me to relieve my burdens.

And, eventually, I want to deeply thank Maria Grazia, whose love always give me the strength to deal with everything the life throws at me, and without her support I am sure that I would not made this far.

[1]Whose name is actual Giuliano, but this is another story...

Contents

List of Acronyms

The following acronyms are used throughout the thesis.

CI/CD	Continuous Integration/Continuous Deployment
RTP	Regression Test Prioritization
APFD	Average Percentage of Faults Detected
APFD$_c$	Cost-Cognizant Average Percentage of Faults Detected
PFF	Percentage to First Fault
PLF	Percentage to Last Fault
EET$_r$	Relative Effective Execution Time
UML	Unified Modeling Language
GA	Genetic Algorithm
APTC	Average Percentage of Transition Coverage
APTC$_{\text{diff}}$	Average Percentage of Transition Coverage on Differences
PMX	Partially Matched Crossover
CPX	Churn-Priority Crossover

LoC	Lines of Code
ART	Adaptive Random Prioritization Technique Max-Min
AST	Abstract Syntax Tree
NLP	Natural Language Processing
TK	Tree-Kernel
STK	Sub-Tree Kernel
SSTK	Subset-Tree Kernel
PTK	Partial-Tree Kernel
MTK	Method-Level Tree Kernel Prioritization
QS	Quotient-Set Prioritization
MTK-QS	Method-Level Tree Kernel with Quotient-Set Prioritization
NOP	No-Prioritization
diff	Difference-Based Prioritization
diff-QS	Difference-Based with Quotient-Set Prioritization
JVM	Java Virtual Machine
JMH	Java Micro-benchmark Harness
VCS	Version Control System
AI	Artificial Intelligence

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Discovering the unexpected is more important than confirming the known

George E. P. Box

Testing plays a crucial role across the lifecycle of a software. It aims to verify the application, ensuring it meets its requirements and that all its components behave as intended to reduce the risk of defects in the released software [125]. Software testing is often performed through the design and the implementation of a *test suite*, i.e., a set of *test cases* whose goal is to verify the behaviour of the application, exercising the operation of its functional components in a controlled environment. The test suite is typically stored together with the source code of the main application in a *codebase*, allowing a readily execution of tests in case of needs.

One of the main scenarios in which a test suite needs to be re-executed is when the software undergo *maintenance activities*. These activities cause changes both in the software source code and in its test suite, by adding test cases for new functionalities or modifying existing ones to ensure consistency between tests and source code. However, in software evolution scenarios it is also possible that modifications to the codebase will result in invalidating previously working functionalities, causing a *software regression*. To cope with this issue, a set of activities in the testing phase are carried out by developers to re-validate the application. These activities are collectively referred to as *Regression Testing* [125, 130].

During the re-execution of the testing phase during software evolution, test case failures are valuable, as they can help the developers to locate and correct the faults which may have been introduced in previously working modules. Nonetheless, when the software is operational for long periods and becomes more mature, the number of test cases in the test suite might grow, and its execution may consume a large slice of testing resources, such as time and money. Moreover, some common developmental practices such as *Test Driven Development* [16, 126], involve the generation of a high number of test cases, and may further worsen the scenario by enlarging the test suite even more over time. In these contexts, it is esteemed that the cost of Regression Testing can account over the 50% of the total cost of development [1, 76, 97]. For these reasons, research community puts a lot of efforts in designing practices to reduce the cost of Regression Testing activities.

In last years, software development has been characterized by rapid changes in requirements, and several novel methodologies, such as *Agile* [3, 38], have spread in order to keep the pace with these rapid changes and be more proactive, in order to satisfy the market needs [109]. These methodologies are often characterized by small and frequent increments in software functionalities, to reduce its time-to-market. Moreover, other practices as *Continuous Integration/Continuous Deployment (CI/CD)* [121, 135] and *DevOps* [120], are commonly applied to further reduce the delivery time of a software. These practices aim to automatize the steps in the software lifecycle, from the build of the application to its final deploy, without involving human control. For this reason, the testing phase is particularly important, as it is the only mean to increase the confidence of a defect-free release.

However, it is not always possible to re-execute the entire test suite, usually due to constraints on the time to release the new software version. For example, the *Google* codebase is esteemed to have over 150 millions of test cases, growing linearly with every update (which, on average, occurs every second) [90]. In similar scenarios, the execution of all test cases might require days or even weeks, consuming a huge quantity of computational resources. Furthermore, feedbacks on failing tests, which may help developers to correct defects, are delayed significantly, thus deferring the software release. To mitigate the problem, several techniques to reduce the

costs of Regression Testing have been proposed in literature and employed in practice.

Regression Test Prioritization (RTP) is one of the most applied approaches to reduce the efforts of Regression Testing, when resources allocated for the testing phase are constrained. RTP aims to re-arrange the order of test cases execution in the test suite, in order to maximize the rate at which faults are detected [115]. The test cases are executed in the order defined by this permutation of the test suite, until all testing resources expire. This can provide benefits to Regression Testing activities, increasing the number of faults which can be discovered even if the testing phase is suddenly interrupted.

Obtaining an ideal permutation of test cases in terms of rate of fault detection is indeed unrealistic, as it involves the previous knowledge of which test cases are going to fail *before* their effective execution. In practice, RTP techniques usually employ heuristics to produce an effective permutation, leveraging different *proxy* measures to build a meaningful permutation, such as *test code coverage* [55, 68, 114], historical information of previous test cases execution [72, 81] or model the problem in meta-heuristic frameworks [48, 62, 141].

Changes made to the codebase are one the main causes of software regressions, as these modifications may introduce defects. However, only few RTP studies in literature (e.g., [65, 116]) consider the *code-churn*, i.e., the modification in the source code which have been made by developers during the update to the codebase. Furthermore, these studies often employ a simple representation of source code changes, which may not fully capture modifications made to the complex structures typical of modern programming languages.

RTP is a highly empirical fields, in which the performance of the various proposed techniques are evaluated and compared to other state-of-art strategies. To this purpose, the vast majority of research studies employ a dataset of benchmark projects, in order to extract software evolutionary scenarios which can be representative of real-world software maintenance. These scenarios are typically modelled through a pair of software versions, to simulate the evolution from a *previous* state of the codebase to a new and updated *current* version. Nevertheless, several employed datasets do not present any fault in the execution of their test suite, due to the fact

that in many cases the update to a code repository is performed by developers only after a successful execution of all test cases. To overcome this problem, RTP scholars usually inject artificial faults in the source code, either manually (e.g., [39]) or automatically via *code mutations* (e.g., [82, 83]).

Lately, due the spread of public code repositories and CI/CD pipelines, such as *TravisCI*[1], in which developers can automatically execute all the software phases, approaches of *Mining Software Repositories* have been also employed in academic research to retrieve software projects in their original building and testing environments, along with execution reports of the various phases [133]. This latter information has been used to collect projects to use in RTP studies including real-world faults, such as *RTP-Torrent* [88].

However, datasets employed in literature are often limited, usually providing only the information on which a small set of RTP techniques relies on. In some cases, these datasets presents projects without the inclusion of their source code, but only some aggregate information (e.g., textual reports of test case failures in previous builds). Furthermore, RTP datasets mined from software repositories seldom include the source code of projects, only providing links to the remote location in which to find the collected versions. In these cases, retrieving the source may not be straightforward, due to repositioning of the codebase or the removal of previous build reports according to the policy of the specific CI/CD pipeline used.

The importance of sharing a common set of data and procedures is paramount in empirical software engineering, in order to generalize results of experiments and derive valid conclusions, limiting threats to the validity of different studies. The scarcity of such data causes researchers of RTP to often struggle to collect a set of benchmark projects which can be employed in the experimentation of new techniques.

My research activities have been focused in filling these gaps. On one side, I designed and evaluate RTP techniques embedding code-churn information in their search for a permutation, with the rationale that software faults and defects are introduced in changed parts of a software source code.

My contribution to the RTP field has been the designing of novel churn-

[1] https://www.travis-ci.com/, visited on 19/01/2024.

based prioritization techniques, in order to state whether the inclusion of information on source code changes can be of any benefit to Regression Testing.

To this purpose, a RTP approach based on the Genetic Algorithm (GA) meta-heuristic framework has been developed, namely *Genetic-Diff*. This technique aims to find a permutation which has a high rate of coverage of the code-churn by employing a novel *fitness function* and *crossover operators*, both leveraging a simple textual representation of code changes. The fault-detection performance of Genetic-Diff has been evaluated on a dataset of software projects with artificially injected faults, employed in several RTP studies [82]. The results of the empirical evaluation have been promising and showed an increase in the fault-detection rates with respect to all employed baselines.

Moreover, I designed two RTP approaches employing a more refined evaluation of the extent of changes in the source code. These techniques leverage Tree-Kernel (TK) functions to evaluate the structural similarity of source code using a meaningful tree-based representation through *Abstract Syntax Trees*. The first technique, Method-Level Tree Kernel Prioritization (MTK), uses this information as a basis to drive the process of test suite permutation, in order to give higher priority to test cases covering higher extents of changed structures. The second technique, Method-Level Tree Kernel with Quotient-Set Prioritization (MTK-QS), is an evolution of the MTK approach and is designed to re-arrange the permutation to increase the rate of coverage of changed code parts. Both techniques have been empirically evaluated on an extension of the dataset employed for Genetic-Diff, to compare their fault-detection rate and execution time performances with several state-of-art baselines. The experimentation showed that MTK is not better than any of the considered techniques, while MTK-QS had significantly higher performance in terms of fault-detection rate and in terms of execution time in almost all considered scenarios.

I also contributed to the RTP field through the collection of *ReCover*, a novel dataset which includes several Java projects with real-world faults, and equipped with a wide variety of useful information, such as coverage reports and full source code. The dataset, namely *ReCover*, includes 114 evolutionary scenarios on 228 different software versions for 28 open-source Java projects, and has been automatically mined from CI/CD pipelines

and code repositories. It has been designed with the goal of recovering missing information from popular collections used in RTP studies. The dataset has been employed in different applications, such as one of my study in *Software Process Metrics* field, investigating the effectiveness of Artificial Intelligence (AI)-based fault-proneness metrics [7], and a preliminary empirical evaluation of several RTP techniques on the projects contained within.

Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 introduces background concepts and definitions for RTP, with the description of common employed metrics to assess the performance of prioritization strategies, and an overview of various techniques proposed in literature, divided by the type of information they leverage.

Chapter 3 presents the *Genetic-Diff* prioritization technique, a novel meta-heuristic approach based on GA framework, which embed code-churn information in its search for a permutation. The chapter provides some background details on GA strategies applied to RTP, and shows the empirical study to assess the effectiveness of Genetic-Diff. The technique and the empirical evaluation have been proposed in my paper [5].

Chapter 4 describes two novel RTP techniques presented in my paper [8], currently in production. The techniques are driven by a more refined evaluation of extents of changes between two versions of a software source code by the means of TK functions. The chapter exhibit contextual information on TK, and details their usage in the proposed technique, along with the empirical evaluation performed to evaluate their performances.

Chapter 5 introduces the *ReCover* dataset, containing heterogeneous open-source Java projects with real faults to support RTP research and published in my paper [6]. The chapter provides a detailed discussion on the dataset collection process, and insights on its characteristics. Moreover, it presents the results of the empirical evaluation of several RTP techniques on the *ReCover* dataset, along with analysis and discussion of such results, both in general and related to sampled specific scenarios.

Chapter 6 summarizes the topics and the findings in this thesis, and presents a discussion on future research lines to pursue.

Chapter 2

Background and Related Work

You can do anything, but not everything.

David Allen

Nowadays, RTP is one of the most widely-used approaches to ease a software's *Regression Testing* state. RTP strategies are applied during the maintenance phases of a software, when a new version should be delivered and constraints on testing resources do not allow the execution of the entire test suite. For this reason, RTP techniques re-order the test-cases in the test suite according to some criteria, and their ultimate goal is typically to schedule the execution of failing tests earlier than the others.

This chapter introduces Regression Test Prioritization and different techniques which have been proposed in literature to cope with it. The first section presents different approaches to ease Regression Testing efforts, and a formalization of the Regression Test Prioritization problem. The second section describes some common prioritization metrics employed in RTP studies to evaluate the effectiveness of different techniques. The last section shows different state-of-the-art techniques which have been proposed in literature to tackle the RTP problem.

2.1 The Regression Test Prioritization Problem

This section introduces some approaches to reduce costs of Regression Testing, including Regression Test Prioritization, and highlights their differences and applicability. It then describes a formalization of the RTP problem and propose a discussion about the possible goals of prioritization, remarking the general intractability of its objectives in several practical contexts.

2.1.1 Approaches to Reduce the Cost of Regression Testing

Automated testing is the primary procedure to check the validity of a developed software. During software evolution, automatic testing has a two-fold purpose: it should guarantee that new requirements are satisfied by the implemented functionalities, and it should also ensure that changes in the software have not adversely impacted previously working functionalities by introducing faults in these modules. Verifying this latter property is the main concern of Regression Testing activities [137].

A naive approach to Regression Testing consists in the blind re-execution of all test cases which were designed for the previous version of the software and have not been deleted by developers (e.g., because the changes in the specifications made them obsolete), with the goal of checking if some of them result in failures. The failure of a test case is often caused by the presence of one or more faults in the software, and failing test cases can be used by developers to locate and correct the faults. Although this naive *retest-all* approach guarantees that all failing test cases are executed, it does not scale well with the growth of the test suite. In fact, as the software becomes more mature and undergoes to several evolutionary steps, the number of test cases in the suite tends to grow larger, and their exhaustive execution could be remarkable long to complete, possibly taking even days. In several scenarios, this cannot be feasible, as it can highly impact the time-to-market of the software or exceed resources allocated for the testing phase.

Several approaches to reduce the cost of Regression Testing solve this issue by executing only a subset of all test cases in the test suite. In particular, three approaches are commonly researched and applied in many practical cases: *Test Suite Minimization*, *Regression Test Selection* and

Regression Test Prioritization.

Test Suite Minimization or *Reduction* techniques [56, 34] aim to reduce the number of test cases in a test suite, driven by the rationale that some test cases become obsolete or redundant after several software evolutionary steps. This could be due to the fact, for example, that the testing of newly-added functionalities can subsume the testing of other subfunctionalities, directly or indirectly called by the software modules which implement the new requirements. Test Suite Minimization tries to produce the minimum subset of the test suite which optimize some arbitrary adequacy criteria, such as code coverage [115]. After the application of such reduction approaches, the size of the test suite will be actually reduced from that point onward.

Regression Test Selection approaches [129] produce a subset of the test suite which will be used in the Regression Testing phase, discarding all other test cases for that particular evolutionary step. The test cases to retain are selected according to some criteria, such as code coverage, test execution history, software quality measures, or by the combination of several of these metrics at once [93, 91]. The cardinality of the subset of selected test cases could be defined *a-priori*, e.g., relative to the amount of resources available for the testing phase, or established according to some target properties the subset should meet (e.g., its overall code coverage or the entity of changes in the code [22]). Differently from Test Suite Minimization, Regression Test Selection techniques do not permanently reduce the number of test cases in the suite, as discarded test cases can still be selected to test subsequent evolution of the software.

Regression Test Prioritization strategies adjust the order of execution of test cases, producing a *permutation* of the test suite [115, 96]. This permutation should provide a more meaningful ordering of test cases, typically to discover faults in the software as soon as possible. Unlike selection, RTP approaches do not fix *a-priori* the number of test cases which are going to be executed, but the test cases are executed in the specified order until the testing resources are depleted, interrupting the testing phase.

Regression Test Minimization and Selection can potentially produce significant flaws in the fault-exposure capabilities of the testing phase [115, 96]. Some studies (e.g., [113]) detected that the capability to expose failures in the software can be severely compromised when applying

Test Minimization approaches. These studies showed that although the costs to perform regression testing are reduced due to the minimization of the test suite, the loss of fault-detection potential after minimization can widely vary depending on several factors, thus leading to an increase of the risks to deploy defective software. Regression Test Selection approaches can also exhibit a loss of fault detection capability for the subset of test cases. Some Minimization techniques are defined as *safe* [111], in the sense that no fault-exposing test case is discarded. The condition to ensure safety, however, is not always granted and it is not generally possible to avoid the potential loss of fault-exposing tests in the minimized subset [112, 115].

The main cause of reduction of fault-exposing capabilities of Regression Test Minimization and Selection is due to the fact that they discard test cases *before* the effective execution of the testing phase. This is particularly true for Regression Test Minimization approaches, which permanently remove test cases from the suite, with the risk of deleting tests which can be redundant for the current evolutionary step but may be useful to exhibit faults in subsequent evolution of the software. On the other hand, for Regression Test Selection, the produced subset can be constrained by the criteria driving the selection, and therefore have a higher cost (e.g., if some overall quality measure on the resulting subset should be guaranteed) or on the contrary have reduced fault detection capabilities (e.g., if the constraint on the size are driven by resource limits). For these reasons, Regression Test Prioritization should be preferred in several scenarios [115]. Indeed, as there is no limit on the number of test cases to be executed when applying prioritization approaches, the benefits for the testing phase can be maximized even if the execution of tests is abruptly interrupted.

2.1.2 A Formal Definition of RTP

One of the first and most general formalization of the Regression Test Prioritization problem can be found in [115]:

Definition 2.1 (Regression Test Prioritization Problem) *Let TS be a test suite and $\Pi(TS)$ be the set of all the permutations of its test cases.*

Given an award *function* $f : \Pi(TS) \rightarrow \mathbb{R}$*, find the permutation* P *such as:*

$$\forall P' \in \Pi(TS),\ f(P) \geq f(P')$$

that is, the permutation P *optimizes the award function* f*.*

It is possible to employ different award functions f, according to the specific goal which a prioritization approach aims to achieve. The formulation of the award function allows to measure quantitatively this goal.

Typically, the main goal of prioritization is to *discover faults as early as possible*, leading to an award function which maximizes the rate at which faults are detected. However, producing a permutation which maximizes the rate of fault discovery results in an *undecidable* problem, as it would provide a solution to the *Halting Problem* [115]. In fact, optimizing the rate of fault discovery supposes the prior knowledge of which test cases are going to fail before their are actually executed. Hence, it needs to know the output of the execution of a program (specifically, a test case) without executing it. A more manageable goal function measures the *rate at which test cases cover code units* in the produced permutation. This goal function is based on the rationale that the quicker the source code is covered, the higher is the chance to exhibit faults in the software. Using this goal function, however, makes the resulting problem NP-hard, as it is possible to reduce the *0/1 Knapsack* problem to it [115, 4]. In several practical cases, sub-optimal heuristic approaches are applied. These approaches often employ one or more *proxy* metrics (e.g., code coverage information, software metrics, test execution history) as contextual information to reorder test cases in the permutation, hoping to achieve high fault-detection rates.

2.2 Metrics to Evaluate RTP Effectiveness

The Definition 2.1 introduces an award function f which is optimized by the permutation of the test suite returned by a RTP approach. The general definition of f can take into account several qualitative goals, and a specific function can be designed accordingly to quantify these goals. The function f is, in this sense, an *evaluation metric* for a permutation [115], giving insights on how much a permutation is effective towards the

selected goal. Due to the generality of f, it is typically not possible to find the permutation which optimizes the award function, and different techniques employ various heuristics to find good sub-optimal permutations. For this reason, the methodology involved in RTP research is highly empirical, and effectiveness metrics play a crucial role in the experimentation. RTP researchers typically use these metrics to compare different approaches, and to state which technique can produce more benefits to testing activities.

The remainder of the section presents the evaluation metrics that are commonly employed to assess the performance of permutations toward specific goals.

2.2.1 Average Percentage of Faults Detected

The most widely-used metric to assess the fault-detection performance of a permutation is the Average Percentage of Faults Detected (APFD) metric [115, 30, 96]. The APFD measures how quickly a permutation can discover faults. It is defined as follows [41]:

Definition 2.2 (Average Percentage of Faults Detected) *Let P be a permutation of a test suite with n test cases, and let m be the number of faults discovered by executing the entire test suite. The* Average Percentage of Faults Detected *for P is:*

$$\text{APFD}(P) = 1 - \frac{\sum_{i=1}^{m} \text{FT}(i)}{n \cdot m} + \frac{1}{2n}$$

where $\text{FT}(i)$ is the index of the first test case that exhibit the i-th fault in the order inducted by P.

The APFD value is in the range $]0, 1[$. The closer the $APFD$ value of the permutation is to 1, the higher is the fault-detection rate for that permutation.

Definition 2.2 arise from a geometrical interpretation of APFD. Given a permutation of the test suite, it is possible to consider the plot which poses on the x-axis the fraction of the test cases in the order of the permutation, and on the y-axis the percentage of faults detected. A point (x, y) on this plot states that the execution of the fraction x of the first test cases in the

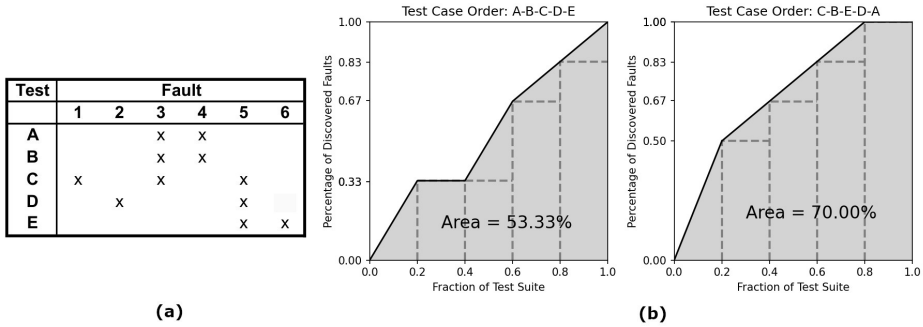


Figure 2.1. An example of the APFD metric. (a) The fault matrix for the example. (b) The test suite fraction/percentage of detected faults plot for two different permutation of test cases. The APFD value is the area beneath the curve in the plots.

permutation discovers the percentage of faults y. Starting from this plot, the APFD value for the permutation is represented by the area underneath the resulting curve.

Figure 2.1 shows a geometrical example of APFD evaluation. Figure 2.1a presents the *fault matrix*[1] for a test suite of $n = 5$ tests (from A to E) and $m = 6$ faults (from 1 to 6). Figure 2.1b presents the test suite fraction/percent of detected faults plots related to two permutations of test cases. The leftmost permutation needs to be executed completely to exhibit all faults, while the rightmost one needs just the first four test cases (i.e., 80% of the entire suite). Furthermore, the first permutation schedules the test case B in the second place, which does not contribute to discover any new fault. A curve is interpolated according to the discrete values (represented by dashed boxes in the figure) of the test suite fraction and faults percentage. As it can be seen, the rightmost permutation has a better fault-discovery rate, expressed by the area underneath the interpolated curves. As APFD measures this area, the APFD values of the two permutations are 0.53 and 0.70, respectively.

To derive the formula in Definition 2.2, consider the plot in Figure

[1] A *fault matrix* is a Boolean matrix $M \in \{0,1\}^{n \times m}$, where n is the number of test cases and m is the number of faults. The entry $M_{i,j}$ is equal to 1 if the test case labeled with i discovers the fault j, otherwise it is 0. In Figure 2.1a, values equal to 1 are denoted by a '**x**' character.

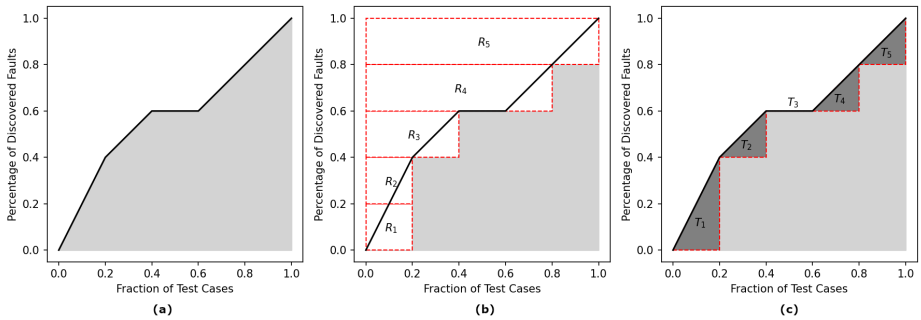


Figure 2.2. Derivation of the APFD formula in Definition 2.2, with a test suite of $n = 5$ test cases and $m = 5$ faults.

2.2a. TheAPFD is equal to the area beneath the curve, which needs to be evaluated. It is possible to note that the whole area of the plot is 1, as it is the area of a square with side length equal to 1. From the whole area, it is possible to remove the area of the m rectangles $R_1, \ldots, R_m$, in order to evaluate the shaded region in Figure 2.2b. Each rectangle R_i has height equal to $1/m$, as it represents the contribute of each fault to the total percentage. To evaluate the width of the rectangle R_i, note that it is equal to the fraction of test cases needed to be executed to uncover fault i, i.e., $\frac{\mathrm{FT}(i)}{n}$, with $\mathrm{FT}(i)$ being the index of the first test case which discovers this fault. Hence, each rectangle R_i has area equal to $\frac{\mathrm{FT}(i)}{n} \cdot \frac{1}{m}$. Adding all rectangles R_i together (with $i = 1, \ldots, m$) provides the summation term in Definition 2.2.

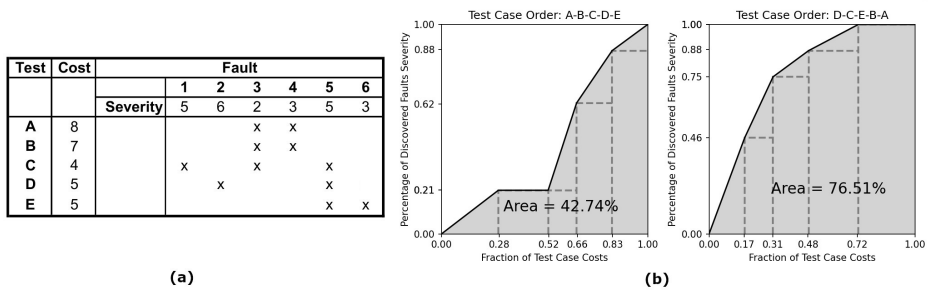
To complete the area beneath the curve, there is the need to add all the right triangles which were excluded while removing the area of rectangles. Figure 2.2 shows the area of these triangles T_i (denoted with darker shades in the figure). At first, note that there is a triangle for each test case, as they are related to the horizontal shifts in the plot. Note that triangles can also have area equal to 0 if the related test case does not contribute to uncover any previously unexposed fault (e.g., the triangle T_3 in Figure 2.2c). To evaluate the area of any triangle T_i, it is possible to see that its base has length equal to $1/n$, as it is precisely an increment in the fraction of executed test cases. If the height of T_i is equal to h_i, the area can be evaluated as $\frac{1}{2} \cdot \frac{1}{n} \cdot h_i$. The total area of all triangles T_i, with $i = 1, \ldots, n$, is

equal to $\sum_{i=1}^{n} \frac{h_i}{2n} = \frac{1}{2n} \sum_{i=1}^{n} h_i$. However, the last summation equals 1, as the sum of all heights precisely provides the 100% percentage of all faults. Thus, the area of all triangles is equal to $\frac{1}{2n}$, which is the last additive term in Definition 2.2.

The APFD metric represents the most common goal of prioritization approaches, as the rate of fault-detection typically measures the effectiveness of a permutation in practical scenarios. In fact, as in RTP the number of test cases that can be executed is not set *a-priori*, the testing phase can be interrupted at any point when testing resources expire. How rapidly a permutation can discover faults can indeed be a useful measure to assess the effectiveness of a RTP technique. However, APFD has some limitations: on one side, it does not provide information on the amount of resources that can be saved by the prioritized permutations, which is often useful in practical scenarios; on the other hand, it assumes that all faults have the same impact on the software and that all test cases have the same, uniform cost of execution [42]. Although these limitations, APFD is often employed in literature as the primary measure to evaluate the effectiveness of RTP techniques.

2.2.2 Cost-Cognizant Average Percentage of Faults Detected

The APFD metric can be generalized, in order to allow it to manage scenarios in which faults can have different impacts and the cost to execute each test case can vary. This more general metric is the Cost-Cognizant Average Percentage of Faults Detected (APFD_c) [42]. This metric extends the APFD function given in Definition 2.2, taking into account different weights both for faults and for test cases. The weight assigned to each fault is a measure of the *severity* of the fault, and it is typically based on the assumption that different faults have different impacts on the quality of the application. For example, a graphical fault in the user interface can be annoying to an user, but does not hinder application functionalities; on the other hand, a fault in the implementation of crucial business logic can severely compromise the software. Weights assigned to test cases are used to quantify the *cost* of the execution of a test. This cost often refers to the actual execution time of a test case, or the price of the infrastructure for that test (e.g., if the testing environment is *cloud*-based and priced by usage).



Test	Cost	Fault						
			1	2	3	4	5	6
		Severity	5	6	2	3	5	3
A	8				x	x		
B	7				x	x		
C	4		x		x		x	
D	5			x			x	
E	5						x	x

Figure 2.3. An example of the APFD_c metric. (a) The fault matrix for the example, specifying costs assigned to test cases and severity of faults. (b) The test suite cost fraction/percentage of detected fault severity plot for two different permutation of test cases. The APFD_c value is the area beneath the curve in the plots.

In scenarios where faults can have different severity and the cost of test cases can significantly vary, APFD can not reflect the effective goal of prioritization. In these cases, it is possible to employ the APFD_c metrics, defined as follows [42]:

Definition 2.3 (Cost-Cognizant APFD) *Let P be a permutation of a test suite with n test cases with costs $t_1, t_2, \ldots, t_n$. Let m be the number of faults in the software under test and $f_1, f_2, \ldots, f_m$ be the severity score of each fault. The $APFD_c$ of P is:*

$$\text{APFD}_c(P) = \frac{\sum_{i=1}^{m}[f_i \cdot (\sum_{j=\text{FT}(i)}^{n} t_j - \frac{1}{2}t_{\text{FT}(i)})]}{\sum_{i=1}^{n} t_i \cdot \sum_{i=1}^{m} f_i}$$

where $\text{FT}(i)$ is the index of the first test case discovering fault i in the permutation.

As APFD, also the APFD_c metric has a similar geometrical interpretation. In fact, it is the area beneath the curve in the fraction of test case costs/percentage of severity of faults discovered plot. The x-axis of this plot represents the cost of executing the test cases in the permutation up to a particular fraction of the test suite, while the y-axis displays the percentage of the total severity of faults discovered by a certain number of test cases.

Figure 2.3 presents an example of the $APFD_c$ geometrical interpretation. Figure 2.3a shows the *fault matrix* for the example, specifying the costs of test cases and the severity of all faults. Figure 2.3b presents the plot of two permutations. The leftmost permutation schedules the test case A in the first position: its execution accounts for a 0.28 fraction of the total cost of the test suite, and discovers two faults (i.e., *3* and *4*) for a 21% percentage of the total severity of all faults. The second test case B increments the fraction of the total cost to 0.52, but it does not contribute in increasing the discovered fault severity percentage, as it does not expose any new fault. Executing the subsequent test cases produces an area of 0.42, which coincides with the $APFD_c$ value of the permutation. On the other hand, the rightmost permutation schedules the test cases in a more meaningful order, and in particular it can be seen that the execution of the first 3 test cases (accounting for the 48% of the total suite cost) discover faults for the 88% of the total severity. The $APFD_c$ metric on this latter permutation is 0.76, showing an increased rate of fault severity-detection.

The main difference between the plot underlying the APFD metric (in Figure 2.1) and the one used to evaluate $APFD_c$ (in Figure 2.3) can be seen in the increments on the axes. The APFD plot presents uniform increases on the x-axis, while these increments on the $APFD_c$ plot are weighted by factors which are proportional to cost of test cases. Furthermore, all increments on the y-axis in the APFD plot are only proportional to the number of faults discovered (i.e., if m faults are present, all increments are proportional to $1/m$). On the other hand, the $APFD_c$ plot presents increments on the y-axis which are proportional to the severity of the discovered faults, i.e., the severity acts as a weight for these increments. Note that if costs and severity weights are equal, the function in Definition 2.3 can be simplified to the classic APFD metric.

The $APFD_c$ metric needs an estimation of the severity of faults and of the costs to execute each test cases. To what concerns test cases, typically their cost is proportional to their execution time, and then normalized with respect to the total time of the test suite complete execution. In specific scenarios, different cost measures can also be employed, as the amount of external resources used by the a test case (for example, in scenarios when test cases require external infrastructures), and these measures can also be combined with test execution times. To estimate fault severity, a common

measure is the *fixing time*, i.e. the amount of time needed by developers to repair the fault.

These estimations for test cases cost and faults severity are however available only after the execution of the test suite or even after the faults are localized, and are not applicable in real contexts. For this reason several studies estimate test costs with their execution times in previous regression testing phases [64], while fault severity is typically estimated through the *criticality* of the requirement or of the module a fault affects [42, 105].

2.2.3 Metrics to Quantify Saved Resources

When evaluating the performance of Regression Testing approaches, it is often interesting to know how many resources can be saved by such approaches. This is particularly true for Regression Test Minimization and Selection, while RTP techniques allow test cases to run until the accounted resources are depleted. It is however useful to evaluate resource-saving related metrics also in RTP, as these metrics can quantify the *latency* of feedbacks of discovered faults, i.e., how rapidly developers could receive notices about the failure of test cases. Although obtaining a permutation which can exhibit faults at higher rates is of utmost importance, quick feedbacks on failed test cases allow an earlier start of the fixing process. In fact, even if the testing phase is still running, developers can focus on fixing the faults discovered so far. To this end, different metrics can be employed and used to complement APFD and APFD_c.

The Percentage to First Fault (PFF) metric evaluates the percentage of test cases in the permutation which needs to be executed until the first fault has been exposed in the software. It can be defined as follows:

Definition 2.4 (Percentage to First Fault) *Let n be the number of test cases in a test suite and let m be the number of faults in the software version under test. Given a permutation P of the test suite, the PFF metric is evaluated as*

$$\text{PFF}(P) = \frac{\min\limits_{1 \leq i \leq m} \text{FT}(i)}{n}$$

where $\text{FT}(i)$ is the index of the first test case in the permutation P discovering the i-th fault.

The resulting PFF value is in the range $]0, 1]$, with lower scores indicating smaller number of test cases needed to be executed to exhibit at least one fault. This metric quantifies the percentage of the permutation execution at which developers will receive the first feedback.

Similarly, the Percentage to Last Fault (PLF) metric is used to measure the percentage of the test suite which should be executed until all faults in the software are discovered at least once.

Definition 2.5 (Percentage to Last Fault) *Let n be the number of test cases in a test suite and m be the number of faults in the software version under test, and P be a permutation of test cases. The PLF metric of P is*

$$\mathrm{PLF}(P) = \frac{\max\limits_{1 \leq i \leq m} \mathrm{FT}(i)}{n}$$

where $\mathrm{FT}(i)$ is the index of the first test in the permutation P discovering the i-th fault.

Note that for all faults, the maximum value of the FT function represents the test case index which discovers for the first time the last remaining fault, as all other faults were already discovered by previous test cases. The range of PLF metric is $]0, 1]$, and a lower score mean that the permutation can uncover all faults more rapidly. This metric is an indicator of how many test cases should be executed before programmers receive feedbacks for all faults in the new software version.

PFF and PLF consider test cases with uniform costs, which is however not the general case. In several scenarios, there is the need to quantify the actual amount of time needed to obtain all feedbacks on faults, in order to have an estimation of the resources saved by a RTP approach. This time should include both the time consumed by the RTP approach to produce the permutation, and the time of execution of all test cases until the last fault is discovered. To this end, it is possible to consider the Relative Effective Execution Time (EET_r) metric of a RTP technique:

Definition 2.6 (Relative Effective Execution Time) *Let S be a RTP strategy, let P be the permutation of the n test cases in the software test suite produced by S, and let m be the number of faults in the software version under test. Given the times of execution of the test cases in the test*

suite $t_1, \ldots, t_n$ and the time T_S consumed by S to produce the permutation P, the EET_r value of S is:

$$\mathrm{EET}(S) = \frac{T_S + \sum_{i=1}^{l} t_i}{\sum_{i=1}^{n} t_i}$$

where l is the index of the first test in permutation P which discovers the last fault for the first time.

In Definition 2.6, $l = \max_{1 \leq i \leq m} \mathrm{FT}(i)$ as previously established in Definition 2.5, and thus the summation adds the execution time of all test cases up to the test discovering the last fault. This metric assesses the time needed to have the feedbacks on all faults, relative to the total time of the test suite. Note that EET_r value can be greater than one due the addition of the strategy execution time T_S. These are the cases in which the execution of a prioritization strategy tends to be too expensive, as it depletes the test resources and reduces the amount of test cases which can be actually executed significantly [59]. Thus, if $\mathrm{EET}(S) > 1$ for a RTP strategy S, then the application of the technique and the execution of the test cases to discover all faults consume more time than the execution of the whole test suite. In this case, there is no benefit in applying the RTP strategy.

2.3 RTP Approaches in Literature

The main goal of RTP activities is to maximize the rate at which faults are detected, and the functional definition of this goal is expressed by APFD or APFD_c. Unfortunately, these metrics cannot be used to drive prioritization approaches, as the information they rely on can be obtained only *a-posteriori*, i.e. after the execution of the test suite. For this reason, RTP techniques employ different criteria, using various proxy measures as intermediate prioritization goals to re-order the test suite. These measures typically include information on previous test execution, which are often available during software evolution, or leverage properties of the version under test to assign priority scores to each test case. In research scenarios, these proxy metrics are used to prioritize the test cases in the suite, then the efficacy of produced permutations is analyzed and compared through

effectiveness metrics.

RTP approaches can be categorized according to the type of information they take into account to drive the prioritization. This section provide an overview of the different types of techniques which have been proposed in the literature.

2.3.1 Coverage-based Approaches

Coverage-based RTP approaches leverage code coverage information to drive the re-ordering of the test suite. The rationale of such methods is founded on the fact that the number of code units exercised by a test case might be used to predict its likelihood to discover faults in the software. The coverage information of test cases in the version under test is typically approximated by the coverage reports for testing activities of previous versions. The coverage information can be exploited at different levels of granularity of code units, e.g., statement coverage [114], branch coverage [55], conditions/decision coverage [68], and method/function coverage [42]. Typically, the desired property to maximize should be the *rate* at which code units are covered by the test cases, but an optimal solution cannot be easily found due to the intractability of the RTP problem using this goal.

Two widely-used coverage-based strategies are employed in practice: the *total* and the *additional* strategies [114, 115]. The *total* strategy assigns to each test case a score equal to the number of code units which are covered by the test, then re-arranges the test cases in descending order of these evaluated scores. The flaw of this technique is that it doesn't take into account which code units have already been covered by previous-scheduled test cases, so the effective coverage rate can be slowed down. Nonetheless, the total technique has a high degree of efficiency and in practice produces acceptable results. The *additional* strategy aims to resolve the redundancy issue of the total strategy. It selects test cases according to the increment on the covered code units with respect to the tests which were already scheduled. The additional strategy typically provides better results than total, but its high execution cost in updating the sets of still uncovered code units makes this strategy not suitable for all applications. To cope with this issue, [54] developed a unified approach, combining both total and additional strategies.

More recently, [63] proposed a novel coverage criterion called *code com-*

binations coverage, which weights test cases according to the various combinations of code elements covered by them. [21] have also proposed meaningful combination of standard test coverage with code change (i.e., *churn*) information. In this study, the authors considered procedure-level test coverage information and devised a strategy focused at covering the procedures that were affected by changes in the new version of the software.

Trying to maximize the rate at which code units are covered can however results in permutations with low degree of fault-detection capability. [55] investigated this phenomenon by designing a technique which maximizes the rate of code coverage, using *integer linear programming*. The technique has been compared with an *ideal* permutation, maximizing the APFD value. The results showed that even the permutations obtained with this optimal technique, maximizing the rate of covered code units, exhibited low scores in terms of fault detection-capabilities. For this reason, solely relying on code coverage could generally provide a good solution in terms of fault-detection rate, and thus other information should be considered.

2.3.2 Churn-based Approaches

The *Code Churn* represents the set of changes in the source code made by developers, when the software migrates from a *previous* version to a *new* one. Some RTP techniques leverage the code churn to drive the prioritization, with the rationale that changed parts in the code are more likely to cause software regressions. These techniques analyze the source code to obtain meaningful information on changes, then prioritize the test cases according to the amount of change they exercise. As all approaches based on changes between software versions, churn-based techniques need to be applied upon the actual release of the software version and could not be pre-evaluated in advance. However, churn-based approaches have a typically reduced cost with respect to other techniques as they solely focus on the changes made to the software, and usually does not consider the unchanged parts of the source code.

The study in [65] presents a technique to prioritize test cases in multi-thread testing scenarios. The approach employs heuristics to find *impacted code points* in the changes between the previous and new version, and then prioritize test cases according to how many threads generated by a

test case exercise the impacted points. The RTP problem has also been modeled as an *Information Retrieval* problem [116]. In this approach, a document collection is extracted from the source code of test cases, and the differences between two program versions is modeled as a *query* on the collection. Differences are extracted using a standard *diff* utility, and then tokenized to create the query. The query results are then produced using different well-established information retrieval techniques (e.g., *TF-IDF* [117] or language modeling [108]). However, to apply this approach a pre-processing phase of the source code of all test cases has to be performed in order to construct the indexes of the document collection. This phase can be time-consuming when the test suite is altered during the software evolution, as it needs the update of the indexes and, if the test suite grows over time, it could not be guaranteed that this process is performed readily. Furthermore, in CI/CD scenarios with frequent updates to the codebase, the updating phase might be triggered again without the previous execution is over.

To the best of my knowledge, few studies in literature focus on a meaningful evaluation of changes, limiting to use heuristics or simple textual tools to analyze source code changes. Furthermore, code churn information typically employed in RTP approaches considers just the amount of change in the software, rather than the effective impact on the software's behavior a change could cause.

2.3.3 Model-based and History-based Approaches

To alleviate the non-trivial costs of collecting test coverage information [20], some researchers have proposed model-based prioritization approaches [75, 15, 74]. These approaches are based on the creation of a *system model*, i.e., a simplified abstraction of the system, and the prioritization leverages the properties of such constructed model. Chief benefit of this kind of approaches is a reduction of the overhead to gather auxiliary information due to this simplification.

An early approach [15] proposed the exploitation of Unified Modeling Language (UML) diagrams in software design model, to automatically generate test cases and to prioritize test cases. In particular, UML diagrams highlighting the dynamic aspect of the model were employed, such as *sequence* and *statechart* diagrams [44]. A RTP strategy which models the

system under test via *Extended Finite State Machine* has been proposed in [75]. This approach appoints higher priority to test cases covering more state transitions in the model, focusing on the software parts which have been updated upon the new release. This strategy has been extended in a subsequent study [74], by employing several heuristic to drive the prioritization. Although these approaches have been found to be promising also in improving early fault detection [73], the cost to build and update a suitable underlying model could not be affordable in all scenarios, and are typically more advisable for *Safety-Critical Systems* [18]. Other model-based techniques leverage *System Dependency Graphs* and *Static Call Graphs* to build the underlying system model [144, 103, 89]. The approaches approximate test coverage information with paths in these static graph models, and prioritize tests cases accordingly.

History-based prioritization techniques leverage project historical information and try to predict which test cases can show failures upon the new release. Typically, these approaches exploit the software history to obtain information on test cases which failed in previous executions [72, 105]. In addition, [81] further refined these intuitions by proposing a prioritization strategy that does not treat all historical information uniformly, but weights the importance of historical data according to the age of the information. Another work [101] takes into account the fact that tests evolve as well, by defining a class of prioritization strategies that rank tests based on their similarity to the previously failing test cases. History-based approaches are typically less expensive than coverage-based and model-based approaches, as they leverage historical information that can be easily collected, typically through the analysis of the build reports collected during the execution of continuous integration pipelines. However, these techniques may not be well adapted to continuously changing testing environments with frequent changes in code and test suites, as these modifications can lead to outdated or erroneous information about test cases [11].

2.3.4 Meta-Heuristics for RTP

A number of different RTP techniques employs meta-heuristic frameworks to produce the permutation of test cases. These techniques aim to find a *good* permutation in the search-space of all permutations of the test suite. The goodness of a permutation is typically evaluated with respect

to some particular desired properties as a function of the permutation, and this function represents the *objective function* of the different techniques. Such search based approaches are applied when it is unfeasible to find the permutation optimizing the goal function of the prioritization problem.

The study in [79] is among the first exploration of these kind of approaches, and presents a comparison of greedy prioritization techniques with hill climbing and GA approaches. GA have been employed to search for an optimal ordering of test cases, usually leveraging history-based or coverage information [62]. Other studies in the field employed other kind of nature-inspired meta-heuristics, such as *Ant Colony* [48], *Bee Algorithm* [98], *Fish School* [141] and *Particle Swarm* [71].

The major flaw of these heuristic techniques relies in their execution time. Although their iterative nature makes possible to interrupt the execution of such approaches at any time, often the short availability of testing resources does not allow the execution of a reasonable number of iterations. Thus, it is possible that the produced permutations are significantly suboptimal with respect to the goal function, and their effectiveness in RTP can be inadequate.

2.3.5 Machine Learning-based Approaches

More recently, prioritization strategies employing Machine Learning techniques have been investigated [102].

Several techniques employ *reinforcement learning* approaches. These approaches aim to train an agent in order to assign scores to test cases, then the test cases are ranked according to the assigned scores. Different reinforcement learning models have been proposed in literature, including *Multi-Armed bandit* [80], *Artificial Neural Networks* and *Random Forests* [20], and these models are used to implement the underlying agent *policy*. Typically, these models consider high level feature to derive policies, such as complexity metrics or historical information.

Supervised learning approaches typically deem the ranking of test cases in the RTP problem in three different models [20]: i) *pointwise* [124, 66], in which the ranking problem is transformed into an ordinal problem, consisting in the labeling of test cases with ordinals representing their degree of priority, and eventually ordering the test cases according to these scores; ii) *pairwise* [77], in which test cases are considered in pairs and the model

predicts the relative ordering of tests in a pair; iii) *listwise* [27], aiming to produce directly the ranking of test cases without any intermediate labeling. Several machine learning models have been employed in these approaches, such as Neural Networks [124], *Support Vector Machines* [66] and *Recurrent Neural Networks* [57].

Even if reinforcement and supervised learning-based RTP techniques have been proven to be effective, their main issue is the need for a time-consuming training phase on a large amount of data, which is not always possible to collect [107, 102]. In particular, [110] used *XCS*, a rule-based classifier, to implement a policy model and showed that a simpler model can produce comparable results with respect to more complex models, but with lower costs in execution time and training data.

Chapter 3

A Change-Aware Genetic Algorithm for RTP

Breed is stronger than pasture

Silas Marner, George Eliot

Search-based meta-heuristic approaches for RTP rarely rely on code churn information, typically leveraging code coverage or software historical data as the basis to drive the prioritization process. The impact of changes is often neglected, and should be better investigated. To fill this gap, I proposed, with my research team, a novel RTP technique based on the genetic algorithm meta-heuristic, with the peculiarity of embedding code churn information in the search of a meaningful permutation [5]. The main purpose of this genetic technique is to analyze whether code changes can be used to drive the search for permutations in a more effective manner.

This chapter describes a novel change-aware genetic algorithm approach designed for Regression Test Prioritization. The first section provides some background information on the GA meta-heuristic framework, detailing how it can be applied to the RTP problem. The second section presents *Genetic-Diff*, a novel prioritization technique, which leverages code churn information to drive the search for a suitable permutation of the test suite. The third section describes the experimental setting for the experimental evaluation of Genetic-Diff which has been performed. The fourth and last section presents experimental results, along with their

analysis.

3.1 Genetic Algorithms and Regression Test Prioritization

The RTP problem can be modeled as a search of a particular permutation, optimizing some award function, among the set of all permutations of test cases in a test suite. Award functions commonly employed in practice cause the problem to be NP-hard, and the search of an optimal solution is too costly. For this reason, a sub-optimal solution is generally acceptable and meta-heuristic frameworks have been typically employed to perform a more clever exploration of the search space of all permutations.

This section describes the fundamental aspects of the GA meta-heuristic, explaining the core terminology and operators that are employed in this framework. Moreover, it shows the typical modeling of the RTP problem as a GA problem, presenting the encoding and the operators which are commonly used for the task of searching a solution to the problem.

3.1.1 Genetic-Algorithm Meta-Heuristic

Meta-heuristics are a wide variety of techniques which are used to find a sub-optimal solution in complex search spaces. Several meta-heuristic techniques are *nature inspired*, i.e., they mimic behaviours and processes in the natural environment to solve a complex search problem, as this sort of behaviour often inherently exhibits some kind of intelligence [134]. Concerning RTP, GAs are one of the most popular and widely-used meta-heuristics in literature to cope with the problem [12].

GAs base their foundation on the evolutionary theory of the *survival of the fittest*, according to which strongest individual have higher chances to survive [61]. In the GA framework, an *individual* represents an encoding of a solution to the problem at hand, and it is composed by *genes*, i.e., the various parts of the encoded solution. Two individuals differ according to the genes they possess. A *population* is a set of individuals with a fixed cardinality, decided before the execution of the algorithm. A GA starts from an initial population, usually randomly generated, and simulates its evolution during various iterations (or *generations*), based on the fact that

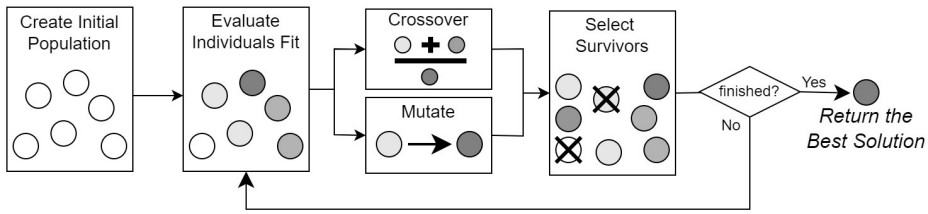


Figure 3.1. Summary of the execution steps in a GA. Darker individuals possess higher values of fitness function.

only the fittest individuals survive and give rise to *offspring*. The fit of one individual is evaluated through a *fitness function*, assessing the goodness of the solution encoded by the individual. The defined fitness function should be suitable for the problem at hand, and the GA searches an individual which could possibly maximize it.

The steps performed by GA are depicted in Figure 3.1. To evolve the population from one generation to another, GA employ two operations: *mutation* and *crossover*. The *mutation* operator randomly alter one or more of the genes of a single individual, producing a brand new solution. Type and probabilities of gene mutations are typically set according to the specific problem. This operation enhances the *exploration* of the search space, allowing the GA to consider different regions of the solution space, with possibly high values of the fitness function. The *crossover* operator, on the other hand, is responsible to breed two *parent* solutions and generate one or more new *child* individuals. To do so, genes in the parents are mixed according to specific criteria, and combined together to obtain the children. Usually, mutating individuals are chosen randomly from the population, with at least one parent in the pair of solutions used for the crossover being an individual with high value of fitness function. The crossover operator enhances the *exploitation* of the search space in regions closer to these fittest individuals.

After applying the crossover and the mutation operator, new individuals are added to the population, increasing its size with respect to the fixed *a-priori* size, and hence a selection step is needed to obtain an updated population with the initially defined cardinality, usually by discarding solutions with low values of fitness function. The remaining solutions are

the *survivors* included in the population for the next generation. The *selector* operator defines the policy to discard individuals. Typical selector operators discard solutions by assigning a probability proportional to the fitness value of an individual, in order to increase the exploration of the search-space. After some designed *finishing criteria* is met (typically, after evolving for a certain number of generations), the GA ends and returns the individual with the highest fitness value found during its iterations.

3.1.2 Genetic-Algorithms Approaches for RTP

To model a RTP problem in the GA framework, the concepts of *gene* and *individual* should be defined accordingly. As solution to the RTP problem is a permutation of the test cases in the test suite, an individual is represented by a sequence, which encodes the permutation. The most common encoding identifies test cases in the suite with a unique integer, and a permutation can be encoded by the sequence of these test case identifiers. To be a valid permutation, the length of the sequence should be equal to the number of test cases in the suite, say n, and each identifier should not appear multiple times therein. In this case, an individual is represented by n ordered *genes*, and the value of i-th gene is the identifier of the i-th test case in the related permutation. Finally, a population is a fixed set of sequences of test case identifiers. Figure 3.2b shows an example of modeling the RTP problem in the GA framework.

The design of a suitable *fitness function* plays a crucial role in the definition of a GA approach. When GA are applied to RTP, one of the common fitness function employed in literature is the rate at which code units are covered by the ordered sequence of test cases, and it is evaluated by the Average Percentage of Transition Coverage (APTC) metric [12, 143, 37]. This metric is often employed as an approximation for the rate of fault detection, with the rationale that a quicker rate of covering the code units can exhibit higher rate of exposed faults. The APTC metric is defined as follows:

Definition 3.1 *Let P be a permutation of a test suite with n test cases, and let m be the number of the considered code units in the software. The APTC of P is*

$$\mathrm{APTC}(P) = 1 - \frac{\sum_{i=1}^{m} \mathrm{FC}(i)}{n \cdot m} + \frac{1}{2n}$$

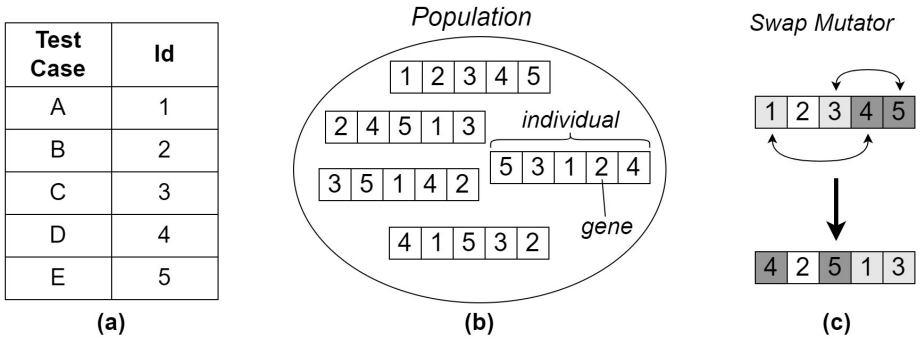


Figure 3.2. Modeling RTP in the GA framework. a) Each test case in the suite is labeled with an unique integer identifier. b) A population of 5 permutations of the test suite. c) An example of the *Swap Mutation* operator.

where FC(i) *is the index of the first test case in* P *which covers the* i*-th code unit.*

The APTC formula has the same form of the APFD metric presented in Definition 2.2. The only difference is in the semantic of the summation. While APFD adds, for each fault, the index of the first test case in the permutation discovering that fault, the APTC summation in Definition 3.1 runs through all the considered code units in the software, adding the index of the first test in the permutation which covers that unit. However, as the APFD is impossible to evaluate before the test have been executed, the APTC metric can be reasonably approximated by using coverage reports from previous software versions in order to link each test to its covered units.

Furthermore, APTC can be geometrically intended as the area underneath the curve depicted by the permutation in the test case fraction/-covered code units percentage plot. The *code units* can be considered at different levels of granularity, and they typically are *statements*, *branch/decision* points, and *functions/methods*, or even multiple criteria at once [37]. During the execution of GA, APTC is evaluated using dynamic coverage reports collected in previous software versions, as, in real context the coverage information of the version under test is unavailable (in fact, test cases have not been executed yet). One of the main bottlenecks of this fitness function lies in the number of code units used to evaluate the function.

Indeed, it is proportional to the size of the software, and larger software tend to have a greater number of code units. Even at a coarser granularity, i.e. function/method, the evaluation of the formula in Definition 3.1 can be time-expensive, as it needs to be evaluated for each individual at every generation, and for this reason it might reduce the testing resources available in the validation phase by just performing the algorithm.

As individuals in GAs for RTP are permutations of the test suite, the employed mutation and crossover operators are constrained to create other individuals which are still valid permutations. That is, the operators should guarantee that no test case is repeated in the produced sequences, thus genes of individuals should have unique values to represent a consistent permutation. Many mutation and crossover operators have been proposed in literature for problems whose solutions are modeled by sequences, and these operators have been applied to RTP as well.

One of the most common mutation operator employed in GA for RTP is the *Swap Mutator* [13, 78, 23]. This operator swaps some genes in an individual to generate a new solution. It guarantees the consistency of the sequence, as it just inverts the order of some pairs of test cases in the permutation, without duplicating or removing any genes. Swap mutator is configured with a tunable probability p, which represents the probability of each test case to be swapped with another one. This probability should be small (usually, $p \leq 0.15$), as higher values cause the swap mutator to collapse toward a random shuffling of the sequence [5]. An example of the swap mutator is represented in Figure 3.2c, where the pairs of swapped test cases are the endings of the arrows. The first and second elements in the swapping pairs are colored with different shades of grey.

Crossover operators manage to combine two parent sequences in one or more children sequences. Genetic-based RTP approaches commonly employ two basic types of crossover operators, both producing two new child sequences: the *Order Crossover* and the *Partially Matched Crossover (PMX)*. Order crossover [35, 143] initially selects a random interval within the sequences. All genes in this interval are swapped between the two parents. As this can lead to duplicate genes, those outside the interval are removed from one of the sequences, and the others are put from the rightmost point of the interval to its leftmost point (as if the sequence was a circular array), to produce the first child. A second child can be con-

structed by repeating the same procedure by swapping the role of the two parents. The order crossover provides a good trade-off between exploitation and exploration, as some parts of a good solution are retained (in the interval), and the other genes are re-arranged (starting from the right end of the interval).

The PMX operator [50, 143] differs from the order crossover operator in the way it performs the re-arranging step. Similarly to order crossover, a gene interval is randomly generated, and the genes within are swapped between the two parents. As these two intermediate sequences can have duplicate and missing genes, the conflicts need to be resolved. To do so, PMX operator builds a map to find substitutes to these conflicting genes, by evaluating the paths of corresponding genes between the sequences. Once the map has been built, conflicting genes outside the interval are swapped according to the map. Figure 3.3 shows an example of the PMX operator. In the first step, the interval is randomly generated (represented by the dashed lines) and the corresponding genes are swapped between the parents. As it can be seen, this leads to a conflict in both sequences: the upper sequence presents two genes with value 1, while the lower contains a duplication of the gene with value 2 (underlined in the figure). To create the substitution map, the gene of value 2 outside the interval is initially mapped to its duplicate inside the sequence, which corresponds to the gene 3 in the upper sequence. This gene is not duplicated in the upper sequence, thus the map proceeds to find the same value in the lower one, which in turn corresponds to the position of gene 1 in the upper sequence. As gene 1 is a duplicate, the ending point of the map for the lower-sequenced gene 2 is completed, i.e., 2 is mapped with 1. The process of the map creation for this conflict is displayed by the path of arrows in the figure. Eventually, genes 1 and 2 are swapped between the two sequences to produce two new valid permutations of test cases.

3.2 The Genetic-Diff RTP Technique

Meta-heuristic approaches to RTP, and GA in particular, usually employ limited information on code churn or does not take it into account at all. Part of my research efforts has been focused to fill this gap, with the goal of investigating whether the embedding of churn information in meta-

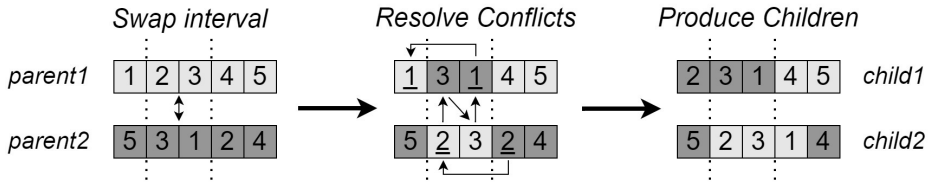


Figure 3.3. An example application of the PMX operator.

Operation	Operator
Fitness Function	APTC_{diff}
Crossover	Churn-Priority Crossover
Mutation	Swap Mutator
Selection	Linear Rank Selector

Table 3.1. The recipe for Genetic-Diff, showing all the operators employed in the implementation of the technique.

heuristic approaches to RTP could provide benefits to regression testing activities.

The *Genetic-Diff* RTP technique has been designed to this purpose. This approach leverages both code coverage and code churn information. Its main idea is that test coverage information should be integrated with details on the *nature* of the coverage, under the assumption that changes in the code might cause software regressions. Thus, higher priority should be assigned to test cases covering more *churned* lines of code. For this reason, Genetic-Diff aims to produce a permutation which has a high *rate of changed covered code units*, i.e., which covers changed elements in the code as quickly as possible. This technique has been designed in the GA framework, as, among meta-heuristic approaches, it provides high flexibility and allows an extensively tailoring of the application to the problem at hand. For this reason, a novel fitness function and a crossover operator have been designed to develop Genetic-Diff, respectively *Average Percentage of Transition Coverage on Differences (APTC$_{diff}$)* and *Churn-Priority Crossover (CPX)*. For mutation and selection operators, Genetic-Diff employs the well-established *Swap Mutator* and *Linear Rank Selector*. An

overview of all operators used in Genetic-Diff is shown in Table 3.1.

3.2.1 Average Percentage Transition Coverage on Differences

$\text{APTC}_{\text{diff}}$ is an extension to the traditional APTC metric, and considers the rate of coverage of changed code units only, rather than the coverage of all code units in the software employed by this latter traditional metric. To investigate if even a naive evaluation of the code churn can produce permutations with good fault-detecting performance, the changes are considered in a *Boolean* fashion: each code unit is either *changed* (i.e., added, removed or modified) or *unchanged*, and the code churn is represented by the set of changed code units between the two software versions. However, as test coverage information is available only for the previous versions, we remove from the code churn all code units which have been added in the new version, and the code churn results in the set of all removed and modified code units. The main advantage of this representation of the code churn lies in its easy evaluation, as it can be readily constructed using even simple textual *diff* tools. Using this representation of the code churn, $\text{APTC}_{\text{diff}}$ can be defined as follows:

Definition 3.2 *Let P be a permutation of a test suite with cardinality n, and let m be the number of code units in the code churn between the two versions of the software. The $APTC_{diff}$ value of P is:*

$$\text{APTC}_{\text{diff}}(P) = 1 - \frac{\sum_{i=1}^{m} \text{FC}(i)}{n \cdot m} + \frac{1}{2n}$$

where $\text{FC}(i)$ is the index of the first test case covering the i-th code units in the churn.

The main difference between APTC in Definition 3.1 and $\text{APTC}_{\text{diff}}$ is the semantic of the summation index i. While in the standard APTC it refers to the i-th generic code units, in 3.2 the cycle runs only on code units which are changed. This reduces the time of evaluation of the function at every granularity: the number of removed and edited code units is generally much lower than the total number of source code elements in the entire software. This is particularly true in modern developmental methodologies,

such as *Agile* or CI/CD, in which the updates to the code base are small and frequent. Hence, $\text{APTC}_{\text{diff}}$ loses its dependence on the size of the software, and is proportional only to the number of the considered code units in the churn.

3.2.2 Churn-Priority Crossover Operator

In order to give more importance to test cases covering changed code, Genetic-Diff employs a tailored CPX crossover operator. It exploits domain information, i.e., the knowledge of which test cases cover code units in the churn, to determine where genes of the two parents should be put in the offspring. To create one child from two parents P_1 and P_2, CPX sequentially scans the parents one gene at a time. That is, for a generic position i, the i-th gene of P_1 is evaluated first and the i-th gene of P_2 is scanned immediately after, before moving to the next position $i + 1$. If an encountered gene has been already inserted into the child, the gene is skipped as its inclusion would introduce a duplicate into the child. Otherwise, CPX chooses the position in which the gene has to be placed into the child by checking whether the related test case covers or not any churned code units. If it covers any element in the churn, it is entitled with higher priority and it is put in the leftmost available empty position in the child. If not, the gene is placed in the rightmost empty position. This process ends when all genes in the parents have been scanned, and there is no more empty space in the child. To generate a second child, the role of P_1 and P_2 is simply switched, and then the CPX operator terminates.

The pseudocode for generating a child with the CPX operator is presented in Algorithm 3.1. The algorithm takes the two parents P_1 and P_2 as parameters. The child is initially empty (line 1), and the variables `frontPos` and `backPos` stores the indexes of leftmost and rightmost empty positions in the child, respectively. Initially, `frontPos` is set to 1 and `backPos` is equal to the last position in any sequence, i.e., its length (lines 2-3). The `while` loop in lines 5-25 performs the operations to place genes in the child. It ends when the child has been completed, that is when `frontPos` become greater than `backPos`, meaning that no empty position is available. The algorithm analyzes the i-th gene of parent P_1 (lines 6-14): if the gene has not been already inserted, it checks if the related test case covers the churn or not (lines 7-13). The auxiliary function `coversChurn`

Algorithm 3.1 The *Churn Priority Crossover* algorithm [5].
Input: P_1, P_2: parent sequences
Output: C: child sequence

1: $child = []$
2: $frontPos \leftarrow 1$
3: $backPos \leftarrow P_1.length$
4: $i \leftarrow 0$
5: **while** $frontPos \leq backPos$ **do**
6: **if** $P_1[i] \notin child$ **then**
7: **if** coversChurn($P_1[i]$) **then**
8: $child[frontPos] \leftarrow P_1[i]$
9: $frontPos \leftarrow frontPos + 1$
10: **else**
11: $child[backPos] \leftarrow P_1[i]$
12: $backPos \leftarrow backPos - 1$
13: **end if**
14: **end if**
15: **if** $P_2[i] \notin child$ **then**
16: **if** coversChurn($P_2[i]$) **then**
17: $child[frontPos] \leftarrow P_2[i]$
18: $frontPos \leftarrow frontPos + 1$
19: **else**
20: $child[backPos] \leftarrow P_2[i]$
21: $backPos \leftarrow backPos - 1$
22: **end if**
23: **end if**
24: $i \leftarrow i + 1$
25: **end while**
26: **return** *child*

is a boolean function responsible to evaluate this condition. If it evalautes to true, the child is put on the `frontPos` index, and the variable is incremented (lines 8-9); otherwise, it is put on the `backPos` position, which

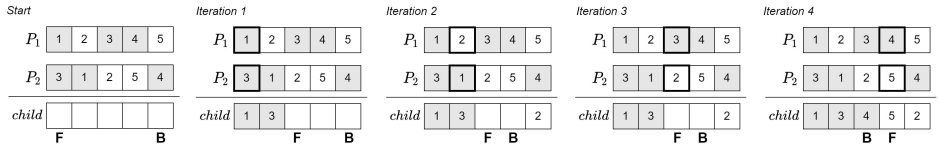


Figure 3.4. An example of the *CPX* operator for a test suite of five test cases. Shaded cells refer to test cases covering elements in the code churn. Bold-edged cells are the elements in the parents which are evaluated in each iteration. F and B represent $frontPos$ the $backPos$ values, respectively.

is successively decremented (lines 11-12). The analysis of the i-th gene in the second parent is done similarly (lines 15-23). After the execution of the `while` loop, the child is eventually returned. Executing Algorithm 3.1 again, swapping the actual parameters P_1 and P_2, provides the second child.

Concerning the algorithm's complexity, note that the `while` loop iterates at most for n iterations, where n is the length of a sequence. In fact, both parents are permutations of the set $\{1, \ldots, n\}$, so the algorithm enters in one of the two `if` constructs in lines 6-14 or 15-23 exactly n times. Every time it enters, either `frontPos` is increased or `backPos` is decremented, eventually making the `while` condition false after at most n iterations (the worst case is actually when the $P_1 = P_2$). Checking if a gene has been already inserted into the child can be implemented in $O(1)$ time and $O(n)$ space using a boolean map, as the number of gene values is upper-bounded by the number of test cases and fixed a-priori. The auxiliary function `coversChurn` can be easily implemented in $O(1)$ time and $O(n)$ space using a map which assigns, to each test case, the value of `true` if that test case covers any element in the code churn, or `false` otherwise. This map can be preliminary created before invoking the entire Genetic-Diff algorithm, as it depends only on test case properties and not on their position in a permutation. As all other statements in the `while` loop take constant time, this implementation of CPX has a complexity of $O(n)$ time and $O(n)$ space.

Figure 3.4 shows an example of the CPX operator, presented in [5]. The example considers a test suite with five test cases, labeled from 1 to 5. Test cases 1, 3 and 4 cover some code units in the code churn in the version

pair, and the two parent sequences P_1 and P_2. At the beginning of the algorithm, the front position is set at index 1, while the back position is 5. The first iteration of the algorithm evaluates the test case 1 from P_1 and test case 3 from P_2. As test case 1 covers the code churn, it is placed in the first position of the child. Test case 3 in P_2 is placed in the next position, as it also covers elements of the code churn. The index of the front position is now 3, while the back position is not changed. In the second iteration, test cases 2 and 1 from P_1 and P_2, respectively, are evaluated. Test case 2 does not cover any element in the code churn, thus it is placed in the last position of the child and the back position is decremented to 4, while as test case 1 from P_2 has already been inserted, it is skipped. In the third iteration, test cases 3 and 2 are considered but are both skipped as they are already in the child. When evaluating test cases 4 and 5, the first is put after test 3, as it covers the code churn, while test case 5, not covering the code churn, takes place behind the test 2. After this iteration, the index of the front position is greater than the index of the back empty position, and the algorithm ends. Eventually, the produced child is the test order 1, 3, 4, 5 and 2. It can be noted that the tests covering code churn have higher priority with respect to those not covering any code churn element.

CPX produces two children in which genes related to test cases covering churn are always prioritized before the others. This can lead to different arrangements of test cases that prioritize the coverage on churn, aiming towards a better exploitation of the search space in terms of $\text{APTC}_{\text{diff}}$. In fact, only test cases covering changed code contributes to this fitness function, and prioritizing these tests before those not covering churned parts of the code has a better probability of producing higher $\text{APTC}_{\text{diff}}$ values. The exploration of the search space is also enhanced by the interleaving of test cases between the two parents, as different orders of churn-covering test cases are produced. In particular, when the size of the test suite grows, it can help to consider several different ordering of test cases. The choice to produce two children also contributes to widen the search space around permutations in which test cases covering changes are scheduled earlier than the others.

3.2.3 Mutator and Selector Operators in Genetic-Diff

To complete the design of Genetic-Diff, the operators for mutating the individuals and for selecting the survivors of the next generation has to be defined. As the crossover operator is generally suggested to be the main contributing operator to find a good solution [143], both mutator and selector operators employed in Genetic-Diff are well-established operators used in GA RTP approaches. The mutator operator picked for Genetic-Diff is the *swap mutator*, defined in Section 3.1.2. This mutator usually guarantees a good amount of diversification in the mutated permutations produced, especially when the number of genes is high (a common case in RTP approaches, as the cardinality of the test suite tends to grow). Thus, the exploration of the search space is widened to more regions in the space, complementing the exploitation capabilities of the CPX operator.

To select the survivors for the population of the next generation, Genetic-Diff leverage the *Linear Rank Selector* [24]. The selector starts from a population obtained at any iteration of the GA, united with all children obtained by crossover and mutated solutions. Firstly, it orders the enhanced set of solutions according to their fitness-value, in ascending order. Then it assigns a probability to each of these solutions, which is proportional only to the rank occupied by the solution in the ordering. Then, it randomly selects test cases according to their probability, until the number of selected elements is equal to the fixed size of the population. Linear Rank Selector evaluates the probabilities as follows:

Definition 3.3 *Given a population of N solutions in increasing order of fitness value, with rank N assigned to the best solution and rank 1 to the worst one, the probability for a solution with rank $1 \leq i \leq N$ to be selected for the next generation is:*

$$P(i) = \frac{1}{N}\left(\eta^- + (\eta^+ - \eta^-) \cdot \frac{i-1}{N-1}\right)$$

with $\eta^- \geq 0$ and $\eta^+ = 2 - \eta^-$, to held the size of the population constant.

In Definition 3.3, the probability to choose the worst and best solutions are equal to $\frac{\eta^-}{N}$ and $\frac{\eta^+}{N}$, respectively. The constraint on η^+ makes η^- the only configurable parameter, and it represents the *reproduction rate* of the

worst individual in the population (typically, $\eta^- \in [0, 1]$). It can be noted that these probabilities are linearly assigned, that is they belong to a line with slope equal to $\frac{\eta^+ - \eta^-}{N(N-1)}$.

The main motivation to employ Linear Rank Selector in Genetic-Diff is due to the fact that, typically, rank selectors are more effective than proportional selectors in GA involving sequences and permutations. As proportional selectors assign a survival probability which is directly proportional to the fitness value of a solution, individuals with very high scores are usually never discarded, limiting the exploration of new solutions [139, 53]. On the other hand, although solutions with high fitness values have a higher rank in the ordering, and thus a greater probability to survive, these probabilities are assigned in a linear and fairer fashion. This helps in reducing the stagnation of the search in narrow regions of the solution space. Furthermore, Linear Rank Selector is also easy to evaluate, and has a time complexity of $O(n \log n)$, where n is the number of test cases [24].

3.3 Methodology and Empirical Setting

To assess the effectiveness of the Genetic-Diff RTP technique, an empirical evaluation has been carried out, aimed at comparing Genetic-Diff to different baseline techniques in terms of fault-detection rate.

This section explains the applied experimental methodology. Firstly, the section details the process of retrieval of the benchmark data, i.e., different subsequent versions of software projects, in which it could be possible to replicate or simulate failures in the source code. Then, it describes the implementation of the Genetic-Diff technique, along with the baseline RTP approaches to compare the performance measure. Finally, it presents the setup of the experiments and the target performance metric.

3.3.1 Data Collection and Subject Projects

Finding subject software projects to perform RTP experimentation is not an easy task. Several datasets in the RTP literature are collected just to be used in the studies where they are presented, and the variety of information used by different techniques often make these dataset incomplete to be used for RTP approaches leveraging on other kind of information.

For example, a small number of datasets provides also the complete source code of contained software projects, while the vast majority of studies uses only high-level metrics on the software. The source code is essential for Genetic-Diff, as the extraction of the code churn would be otherwise impossible. Another issue related to the dataset collection is that publicly available projects seldom provide real faults, as often their versions are published only after potential test failures have already been corrected. As the utmost goal in prioritization is generally the rate of fault detection, it is a common practice in RTP study to use artificial faults, injected either manually or automatically in the benchmark software.

Among several studies in literature, [82] presents a dataset of open-source Java projects with different size and nature, with the full source code and faults automatically injected via *code mutation*. Information on the fault locations is also included in the dataset, making the evaluation of several fault-detection metrics easier. Furthermore, a replication package has previously been published[1], containing several different versions of all subject projects. Each version is furnished with its full source code, coverage reports and the location of all methods in which artificial faults were injected. The software versions employed in the study ranged between 2012 and 2015, and the source code is representative of modern Java language constructs.

From the dataset in [82], three projects have been selected for the empirical evaluation of Genetic-Diff, as they can be representative of software projects with different size and various number of test cases. To mimic software evolution scenarios, four versions of each project have been collected and labeled from 1 to 4 in ascending order of their release date.

Table 3.2 show some statistics for the projects involved in the empirical evaluation of Genetic-Diff and some of their properties. *JOpt*[2] is a small auxiliary library used to parse command line arguments. Even if its Lines of Code (LoC) number is on average around 7k, the number of test cases across the different version is relatively high. *Metrics*[3] is a Java tool used to extract software static metrics used by developers in analysis and debugging. Subject of the empirical analysis is the *metrics-core* package,

[1] But, sadly, it is no more publicly available.
[2] https://github.com/jopt-simple/jopt-simple, visited on 17/12/2023.
[3] https://github.com/dropwizard/metrics, visited on 17/12/2023.

Project	LoC	# Tests	Description
JOpt	7k	427	Library for argument parsing
Metrics	11k	294	Tool to compute software metrics
AssertJ	62k	2419	Assertion framework for testing Java applications

Table 3.2. Subject projects for the empirical evaluation of Genetic-Diff. *LoCs* and number of test cases are averaged between the 4 versions of each project.

which offers the basic functionalities used throughout the entire tool, and it is a medium-small project with less than 300 test cases on average. The last considered subject project is *AssertJ*[4], a framework providing a rich set of assertion used for testing various java application and compatible with several *test runner* utilities. It is a large sized project, with an average of more than 60k LoC. It also provides a test suite with over 2400 test cases on average.

Concerning the injection of faults, the original experiments in [82] used *code mutation* to simulate developer errors and, thus, faults. Code mutation is a process consisting in the modification of the source code by altering some syntactic constructs in order to create unintended behaviours in program functionalities. It is applied in *mutation testing* to assess the quality of a test suite [104]. In the field of RTP, code mutation is often used to simulate real faults, due to not negligible efforts to retrieve software project exhibiting this type of faults. However, typically faults injected via code mutation are good substitutes for real faults, as the detection of mutants is highly correlated with the detection of real faults [70].

[4] https://github.com/assertj/assertj, visited on 17/12/2023

Operator	Name	Description
AOR	Arithmetic Operator Replacement	Replaces a binary arithmetic operator with compatible alternatives.
COR	Conditional Operator Replacement	Replaces a conditional operator with compatible alternatives. Also replaces atomic Boolean conditions with `true` and `false`.
LOR	Logical Operator Replacement	Replaces a binary logical operator with compatible alternatives.
ROR	Relational Operator Replacement	Replaces a relational operator with compatible alternatives.
ORU	Operator Replacement Unary	Replaces a unary operator with compatible alternatives, e.g., `-a` can be replaced with ∼`a`.
LVR	Literal Value Replacement	Replaces a literal with a default value. A numerical literal is replaced with a positive number, a negative number, or zero. A Boolean literal is replaced with its logical complement. A String literal is replaced with the empty String.
EVR	Expression Value Replacement	Replaces an expression with a default value, such as 0 or `null`.
STD	Statement Deletion	Removes specific kinds of statements, such as method invocations, pre/post increment operators and assignments.

Table 3.3. Complete list of all mutant operators provided by the *Major Mutation Framework* and which have been used in the fault-seeding process of the Genetic-Diff experimentation.

The mutant injection process in the evaluation of Genetic-Diff followed the approach previously presented in [82]. Mutants were injected in the fourth and most recent version available of each project, in methods which are present in all versions and have not been deleted. The list of these methods was available in the replication package of [82], and were originally chosen through an analysis of the evolution of the software. The rationale behind the choice to inject faults only in the most recent version is to simulate different incremental steps in the evolution of the project, setting the fourth version as the target version of the prioritization activity. As the precise nature of the mutant (i.e., information on what kind of code modification had been executed) was not available in the original package, the mutants involved in the experimentation of Genetic-Diff is slightly different, but the faults were however injected in the same methods where they were injected in the original study.

To create and inject mutants in the dataset, the *Major Mutation Framework*[5] has been employed. Although different tools to perform code mutation are available in the Java ecosystem (e.g., *PIT*[6] or *MuJava*[7]), the vast majority of them performs the mutation directly on the Java *bytecode*[8], and it is impossible to precisely reconstruct the actual source code mutation. On the other hand, *Major* applies mutations directly in the source code, and supplies a wide variety of code mutation operators (presented in Table 3.3).

Following the methodology in [82], starting from the set of mutants for faulted methods in the last version of each subject software, a collection of 100 *variants*, with different sets of injected mutants, has been created for each version. According to previous studies in literature (e.g., [83]), the number of injected faults does not affect the evaluation metrics typically used in RTP. For this reason, the number of mutants has been set to 5, which is the same number used in [82], as it represents a reasonable value to evaluate fault-proneness metrics. Thus, each variant presents 5 injected mutants, which were randomly selected from the mutant set, in order to simulate faults in the software.

[5] https://mutation-testing.org/, visited on 18/12/2023.

[6] https://pitest.org/, visited on 18/12/2023.

[7] https://github.com/saiema/MuJava, visited on 18/12/2023.

[8] *Bytecode* is an intermediate language produced by the compilation of Java code, which is interpreted and executed by the Java Virtual Machine.

The source code of each variant consists of the original code of the version, with the exception of the methods which are faulted in the variant. For these methods, the mutated code is embedded in their original source code. A fault matrix has then been built for all variants, accordingly to test cases which fail due the injected mutants.

3.3.2 Genetic-Diff Implementation and Baseline Techniques

To perform the experiments, the Genetic-Diff technique has been implemented in Java language using the well-known *Jenetics* Java framework[9]. Jenetics is an optimization library specialized in the definition and the execution of GA, *Evolutionary Algorithms*, *Multi-Objective Optimization*, and many other meta-heuristics. The main strength of Jenetics is a clear separation between core concepts of the GA framework, such as the various operators employed in this meta-heuristic, with the actual execution flow. Furthermore, Jenetics provides several commonly-used operators already implemented, easing the effort to implement new techniques. For these reasons, only the $\text{APTC}_{\text{diff}}$ fitness function and the CPX operator have been implemented from scratch and embedded in the Jenetics framework, and the Genetic-Diff approach has been readily developed. Jenetics also allows a parallel execution of the algorithms in an user-transparent fashion, and with several configuration options. This causes a reduction of the execution times, both in research and practical contexts.

The first point in the setting of the experimentation of Genetic-Diff has been the selection of the granularity of code elements used in its evaluation. To this end, the $\text{APTC}_{\text{diff}}$ used in Genetic-Diff employs the *statement* granularity level. This level of granularity has been typically used in different studies [142, 128] and usually provides permutations with better performances in terms of fault-detection rate [12]. The empirical evaluation of Genetic-Diff is performed at the same granularity. In this setting, the $\text{APTC}_{\text{diff}}$ fitness function measures the rate of coverage of changed statements, and thus the value of m in Definition 3.2 is equal to the number of churned statements in the source code. Another design choice for the experimentation is related to the probability of gene mutation in the Swap Mutator operator. This probability has been set to the commonly

[9] https://jenetics.io/, visited on 19/12/2023.

recommended value of 0.1, in order to have on average a mutated solution with a percentage of 10% of swapped genes. For the parameters related to the execution of Genetic-Diff, the fixed population size has been set to 100 individuals, while the stop criterion considers the number of evaluated generations, and this number has been set to 200. These values were commonly used in several studies which empirically evaluate GA for RTP [79, 82].

In order to perform the empirical evaluation, three baselines RTP techniques have been selected to compare Genetic-Diff with:

- *Total*: the classical and widely-adopted greedy technique which orders test cases according to the number of code units they cover [114]. This is a deterministic technique, leveraging coverage information from the previous software version to approximate the coverage of test cases in the new version under test. In the experimentation, coverage information is considered at the *statement* granularity. This technique has been chosen for the experimentation as it is commonly used in practical contexts, due to its straightforward implementation and its good performances in several applications.

- *Adaptive Random Prioritization Technique Max-Min (ART)*: an adaptive random technique [67], trying to prioritize test cases according to their diversity. Initially, it generates a *candidate set* by randomly selecting test cases until all statements are covered by this set. Then, it iteratively selects, from the candidate set, the test case which maximizes the minimum distance with the partial sequence of tests already selected. To evaluate the distance between two test cases, ART considers the sets of covered elements of the two test cases, and assesses their distance by subtracting their *Jaccard Index*[10] from 1. This approach has been included in the experimental evaluation to compare Genetic-Diff with an RTP technique using a different metaheuristic.

- *Genetic-APTC*: a GA using the classical APTC fitness function, at statement granularity level, using the PMX operator as crossover

[10]Given two sets A and B, their *Jaccard index* is $J(A,B) = \frac{A \cap B}{A \cup B}$. The more $J(A,B)$ is close to 1, the higher the similarity between the sets [85].

and Swap Mutator as mutation operator. This technique has been added to evaluate if there could be benefits in the inclusion of churn information compared to a similar approach which is only based on coverage. To assess a fair comparison with Genetic-Diff, the population size and the number of iterations have been set to the same values, i.e., 100 and 200 respectively.

As a note, the choice of the *Total* strategy over than the *Additional* strategy, as the deterministic baseline for Genetic-Diff, has been motivated by the fact that the application of the latter strategy is often infeasible in practical contexts. Indeed, although the *Additional* RTP technique performs better than the *Total* strategy, it does not scale well with the number n of tests in the application, as its presents an asymptotic time complexity of $O(n^2)$, due to the need to update each not already-scheduled test case with the revised number of uncovered code elements. As the number of test cases typically present in software on which RTP strategies are needed to be applied tends to grow, this quadratic scaling could easily exceed the amount of resources allocated for testing.

3.3.3 Experimental Setup and Evaluation Metrics

The definition of the experimental setting should simulate, and be representative of, real RTP scenario arising in practice, as much as possible. As RTP is applied when a software evolves from a *previous* version V_i to a *current* version V_j, the base of an experiment involves a *version pair*, i.e., the pair (V_i, V_j), with V_j being the version under test. In the Genetic-Diff experimentation, for each project, the version V_j is always the fourth and last version of that project in which faults have been injected, named V_{4F} to distinguish it from its original version V_4. Hence, each version pair has the form (V_i, V_{4F}), with $i \in \{1, \ldots, 4\}$. Pairs with $i \leq 3$ simulate different entity of changes between the previous and the new version, and can be representative of iterative updates to the code base typical in modern *Agile* practices. Version pairs (V_4, V_{4F}) are instead unrealistic scenarios, as the only changes in the code are related to the injected faults. These pairs were also included for evaluation purposes. In total, the experiments involved 12 version pairs.

As the faulted version V_{4F} of each project comes with 100 variants

with different groups of faults, the execution of the techniques has been evaluated separately for each one of the variants, in order to represent up to 1200 different evolutionary scenarios. The data is then aggregated on a version pair basis, to mitigate the effects of possible outliers in the performance evaluation.

All the baseline techniques, along with Genetic-Diff, use coverage information. In order to mimic a real-world scenario, for each experimental version pair (V_i, V_{4F}), the coverage data used by the approaches is collected from the execution of test cases of the previous version V_i. As the versions are numbered by their release date, older versions have older coverage reports, which can be worse approximations of test coverage in the new version. This is also useful to analyze on what extent the freshness of coverage information impacts on the evaluated prioritization techniques.

Primary goal of the experimentation is to assess whether the exploiting of churn information could lead to benefits, in terms of fault-detection rate, to RTP activities. For this reason, the performance metric employed in the study is the APFD metric presented in Definition 2.2, and it is evaluated for all permutations produced by the considered approaches. Genetic-Diff and all baseline techniques have been executed on each variants for all experiments, and their APFD scores have been aggregated for each version pair.

To assess the statistical significance of the evaluation, the *Mann-Whitney* test [84] has been performed on the APFD values, aggregated for each version pair. In particular, the test aims to verify if the APFD achieved by the Genetic-Diff technique is significantly greater than the score of the other baselines. Thus, the *null hypothesis* can be defined as follows [5]:

Definition 3.4 *For each subject project s, version pair v and baseline b, the Mann-Whitney test* null hypothesis *is:*

$\mathbf{H_0}^{s,v,b}$*: The APFD achieved by Genetic-Diff on the version pair v of the subject project s is not greater than the APFD value achieved by the technique b in the same setting.*

To have a high degree of confidence in rejecting the null hypothesis, the threshold for the *p-value* has been set to $\alpha = 0.05$. With *p-values* lesser than this threshold, it is possible to accept the alternative hypothesis,

	AssertJ				JOpt				Metrics			
	$1 \to 4F$	$2 \to 4F$	$3 \to 4F$	$4 \to 4F$	$1 \to 4F$	$2 \to 4F$	$3 \to 4F$	$4 \to 4F$	$1 \to 4F$	$2 \to 4F$	$3 \to 4F$	$4 \to 4F$
Total	0.63	0.66	0.71	0.76	0.91	0.85	0.91	0.93	0.79	0.78	0.75	0.78
ART	0.86	0.89	0.83	0.91	0.81	0.90	0.92	0.93	0.71	0.70	0.72	0.74
Genetic-APTC	0.87	0.87	0.83	0.88	0.89	0.92	0.96	0.97	0.82	0.83	0.84	0.87
Genetic-Diff	0.88	0.90	0.91	0.97	0.94	0.98	0.98	0.99	0.89	0.90	0.91	0.91

Table 3.4. Average values of APFD obtained during the experimentation of Genetic-Diff on the subject projects, aggregated by version pairs considered in the experiment [5].

stating that Genetic-Diff performs better than the baseline in the same setting.

3.4 Results and Discussion

This section present the results of the empirical evaluation of Genetic-Diff for the three selected benchmark projects, and discusses the interpretation of these results, along with threats to validity that could have affected the generality of results.

3.4.1 Results of the Empirical Evaluation

Genetic-Diff and the other baseline approaches have been executed on all the experimental scenarios drawn in Section 3.3.3. For each version pair (V_i, V_{4F}) of each project, the different techniques have been executed on all variants. Then, an APFD value has been evaluated for all the permutations produced by the techniques on each single variant.

Table 3.4 presents the obtained results for Genetic-Diff and all baseline techniques considered in the experimental setting. The table reports a column for each version pair in the experimentation, and one row for every executed technique. Each cell contains the average APFD value of all 100 variants of the version pair on the column for the technique on the particular row.

As it can be seen, the average APFD value of the Genetic-Diff technique is always higher than the results observed for all the baselines. Considering the version pair (V_4, V_{4F}) for all the projects, Genetic-Diff achieves values very close to the APFD upper-bound of 1. This is due to the fact that,

in these cases, all changes between the original version V_4 and its faulted version V_{4F} are the precise locations where mutation faults were injected. Although this is not a realistic case, it is however possible to see from the results that $\text{APTC}_{\text{diff}}$ and CPX cause Genetic-Diff to generally produce permutations in which test cases covering churned statements have higher priority. In all other experiments, i.e. (V_i, V_{4F}) with $1 \leq i \leq 3$, the performance of the Genetic-Diff technique are on average better than the baseline techniques, showing an improvement between 2% and 9% in the APFD scores with respect to the second best-performing technique.

Concerning the freshness of coverage reports in the evaluation of the techniques, it is possible to see the *total* baseline technique significantly improves in the *AssertJ* project as the previous version V_i get closer to the last version. In this project, however, solely relying on coverage information showed poor performances, as Genetic-Diff and all the other baselines produce better APFD values. Indeed, the randomness component in their execution seems to make them less influenced by the freshness of coverage reports. Considering the first version pair (V_1, V_{4F}) of the *JOpt* project, the *total* technique produces a mean APFD which is comparable with the other baselines. However, starting from the pair (V_2, V_{4F}) the trend already seen in *AssertJ* shows again. In the *Metric* project, instead, the performance of the *total* technique shows fluctuations while moving towards versions pairs involving more recent previous versions. This could be caused by project size and test cases number, as the amount of covered parts between versions does not change significantly.

To analyze also the variation of APFD results of all technique, the data has been depicted via the box-plots in Figure 3.5. As it can be seen, the Genetic-Diff technique performs generally better also in terms of medians (the lines inside the boxes) and has also reduced variance in all cases, showing narrower boxes. The trend regarding freshness of coverage reports can be seen as well in this plot, both for *AssertJ* and *JOpt*. On the other hand, the *Metrics* project shows no sensible correlation of the results with the freshness of coverage reports, for the motivations previously stated.

Figure 3.6 presents the results of the statistical significance Mann-Whitney test on the APFD results. According to the *null hypothesis* $\mathbf{H}_0^{s,v,b}$ in Definition 3.4, to each software project s, version pair v and baseline technique b, the corresponding cell shows the *p-value* obtained by the test.

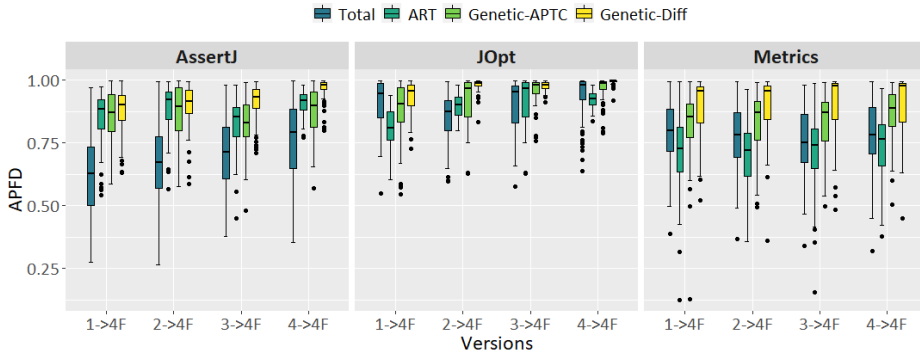


Figure 3.5. Box-plots of aggregated APFD values in Genetic-Diff experiments [5]. Different boxes relates to different techniques and consider the aggregation of APFD values among the 100 variants of each version pair.

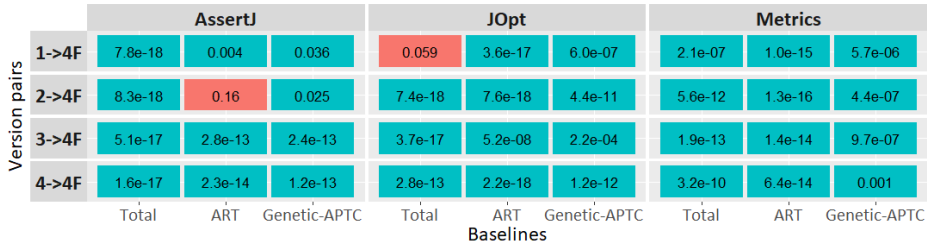


Figure 3.6. Mann-Whitney Significance test for the APFD results in the Genetic-Diff experimentation [5].

When it is possible to reject the null hypothesis with high confidence, i.e., $p < 0.05$, the cells are shaded in green. In these cases, the alternative hypothesis should be accepted, that is Genetic-Diff performs significantly better then the baseline b on the version pair v of project s. On the other hand, if in some setting it is not possible to confidently reject the null hypothesis, the corresponding cell is shaded in red.

The results of the analysis stated that the null hypothesis can be rejected with very high confidence ($p \ll 0.05$), in almost all experimental settings. In these cases, Genetic-Diff performs significantly better than the other baselines. The only two experiments in which the hypothesis cannot be rejected are version pairs (V_2, V_{4F}) of *AssertJ* and (V_1, V_{4F}) of

JOpt. In the former case, the ART technique achieve similar performances (as it can be also seen by the box-plots in Figure 3.5), while, in the latter, Genetic-Diff is not significantly better than the *total* technique, even if In this latter case, however, $p = 0.059$, which is slightly higher than the confidence threshold.

The results of the analysis showed that Genetic-Diff can provide benefits to RTP activities in terms of fault-detection rate, and in almost all cases these benefits are significant. This suggests that the employment of churn information in prioritization techniques can help regression testing activities, and that the code churn can be a valid measure to drive the process of re-ordering test cases, even though it is measured in a simple Boolean manner. For this reason, a further question on which I focused during my research has been to investigate whether more refined approaches for evaluating code churn could be profitably used to guide the construction of a test cases permutation towards increased fault-capability performances, and if these more precise approaches could perform satisfyingly even if not included in more complex frameworks, as the GA meta-heuristic.

3.4.2 Threats to Validity

This subsection presents the threats that could have affected the results of the experiments and their generalizability, following guidelines proposed in [140].

Threats to internal validity

Internal threats to validity relates to confounding variables which limit the confidence in the cause-and-effect association between the experimental conditions and obtained results.

To ensure a fair comparison between all the techniques, they were executed on the same machine, with similar load-conditions, and allowed to each of them the same amount of time for termination. All the techniques used the same input information, i.e., coverage reports, test reports and source code (when needed) at the same level of granularity. In particular, to guarantee a fair comparison between the Genetic-Diff and the baseline Genetic-APTC, the population size, number of generation and common operator probabilities were set to the same values. Also, the

three RTP baseline techniques were re-implemented according to the way they were presented in previous works, and adapted to a common experimental framework. For this reason, it is not possible to exclude some differences between the original implementation and the one used in the empirical evaluation of Genetic-Diff. To reduce this risk, the implementation of some baselines used the original source code, when available, provided by the authors, and were only wrapped within the module of the experimentation pipeline.

Threats to external validity

External threats to validity refers to possible issues which can weaken the generality of findings and results.

A threat to external validity can arise from the set of software projects considered in the evaluation. Three open-source Java projects have been employed in the experimentation, and these projects have been already used in prior RTP researches. The selected projects were chosen to be as heterogeneous as possible, varying both in size (small, medium and large) and in the number of test cases (from around 200 test cases to over 2400). However, the number of projects involved in the evaluation might not be representative of all kinds of Java projects.

Another threat to external validity is due to the usage of artificially injected faults. In fact, as no real-world fault was available for the project, faults were injected through code mutations. However, several RTP studies employed the same kind of faults [82, 83], and the process of fault seeding has been carried out to follow these widely-used methodologies as much as possible. Although these kinds of faults could not be representative of all possible types of errors occurring during development, some studies (e.g., [70]) stated that code mutants could be a suitable replacement to real-world faults in *in-vitro* RTP experiments.

Chapter 4

Tree Kernel Prioritization Techniques

Time spent amongst the trees is never wasted time

Katrina Mayer

Typically, the various RTP techniques proposed in literature do not leverage changes in the source code at all, or they do just to a limited extent. Even when this happens, they often consider only the *textual* similarity of source code between software versions, completely ignoring the complex structures generated by the grammar of different programming languages. Indeed, a natural and more informative representation of the source code is through tree-based structures, such as *Syntax Trees* and *Abstract Syntax Tree (AST)*, which can better capture relations between the various elements composing the code (e.g., a statement in a loop rather than in a branch block), simultaneously highlighting the semantic of these elements.

For this reason, part of my research has been centered on the application of more refined methods to quantify the similarity between source code fragments through their AST representation, in order to employ this information to drive the construction of a test case permutation. Among the different models to evaluate the similarity of tree-structures, I focused on TK functions due both to their flexibility for different operative con-

texts, and to their customizability for the specific problem at hand. Chief goal of this research has been to investigate whether a novel RTP technique based on these more refined measure of code-churn similarity could lead to benefits in regression testing activities. This technique, namely the MTK approach, and its evolution, the MTK-QS approach, have been proposed in [8].

This chapter presents the MTK and MTK-QS techniques to prioritize test cases, along with the results and the discussion of their empirical evaluation. The first section gives background information on TKs and their application in the fields of Natural Language Processing (NLP) and Software Engineering. The second section shows how TK functions can be employed to measure the similarity of the AST representation of source code. The third section presents the novel MTK and MTK-QS approaches to RTP, designed during my research activities. The fourth section describes the methodology and the experimental setting for the empirical evaluation, while, in the fifth section, the analysis of the obtained results is presented, along with a comparison with different widely-used state-of-the-art RTP techniques.

4.1 Tree-Kernels: Background and Applications

This section presents an overview on TK functions. At first, it gives some information for *kernel* functions, and in particular *convolution kernels*, contextualizing TKs in this more general framework. Subsequently, the section describes the family of Tree Kernels, along with the most used specific functions in this class.

4.1.1 Kernel Functions and Convolution Kernels

Kernel functions [29, 49, 60] have been extensively studied in the field of *Machine Learning*. They are capable of evaluating the similarity of two objects through a dot product in a high-dimensional space where the objects are projected as data points. The evaluation of the dot product is performed *implicitly*, i.e., without the intermediate calculation of the data points in this new space, allowing an efficient similarity evaluation in space with very high (or even infinite) dimensionality [28]. Given a set

X, a function $K : X \times X \to \mathbb{R}$ is a *kernel* function if it is *symmetric* and *positive definite*[1] [58].

Convolution kernels [58, 123, 100] are a family of kernel functions which are used to evaluate the similarity between two discrete composite structures, such as sequences, trees or graphs. Convolution kernels consider the *fragments* of the composite structures to evaluate their similarity. Fragments are specific sub-structures, of different types, embedded in the primary structure of the composite object. Each pair of fragments that the two structures have in common contributes to the similarity of the objects, and the total overall similarity is evaluated by adding together the contribution of all pairs of fragments. Different convolution kernel functions are classified according to the type of discrete structure of which they evaluate the similarity and the kind of fragments they consider, e.g., *sequence kernels* [26, 138], *string kernels* [58] or *random walk graph kernels* [136].

4.1.2 Tree-Kernels to Evaluate Similarity of Tree Structures

TK functions are particular convolution kernels which have been specifically designed for tree-based structures and have been widely adopted in many fields in computer science, such as NLP [31, 95] and software clone detection [32]. They can evaluate the similarity of two *ordered labelled tree* structures, taking into account both information of their topology and the role of the nodes within. In fact, ordered trees are a particular kind of tree structures in which the children of each node are expressed by a sequence, rather than a set, in which a node has a precise position. This induced order in child nodes provides details to the tree structure, carrying another piece of information. A labelled tree is a tree structure which highlights the semantic of each node due to *labels*, i.e., value nodes are marked with and which reflect the connotation of the node in the entire structure. An ordered labelled tree has the properties of both ordered and labelled tree.

An example of an ordered labelled tree, from the NLP domain, is depicted in Figure 4.1. The sentence "*The cat eats a fish*" is represented as a tree structure through the rules of the English grammar, and semantic

[1] A function $K : X \times X \to \mathbb{R}$ is *symmetric* if for all $x, y \in X, K(x, y) = K(y, x)$. It is *positive defined*, if for all $n \geq 1$ and any $x_1, \ldots, x_n \in X$, its *Gram* matrix $K_{i,j} = K(x_i, x_j)$ is positive definite, i.e. for all $c_1, \ldots, c_n \in \mathbb{R}$, $\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K_{i,j} \geq 0$.

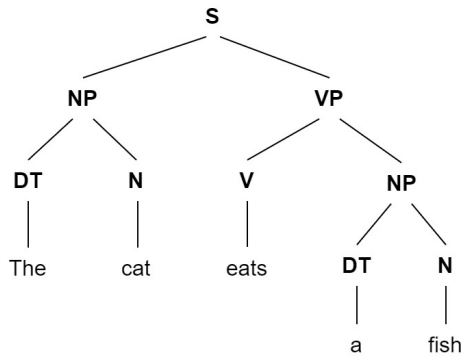


Figure 4.1. An example of ordered labelled tree for the sentence "*The cat eats a fish*".

and syntactic information is highlighted by the node labels. The root node label S tells that the entire tree is a *statement*, and it is composed by a *noun phrase* and a *verbal phrase* (its child nodes labelled with NP and VP, respectively). The noun phrase node is the parent of a DT node, which is referred to as *determiner*, and a *noun* node labelled with N. On the other hand, the VP node is the parent of a verb node labelled with V and another *noun phrase*, representing the direct object of the sentence. The actual words of the sentence are stored in leaves (i.e., *terminal* nodes). This representation preserves the pieces of information of the sentence both in its structure (e.g., allowing to distinguish the subject noun phrase from the direct object thanks to their position in the tree) and in the label assigned to each node, which carries its semantic.

Due to their specialization to operate on ordered labelled trees, TKs have been designed to take into account both the structural properties of a tree, through the usual parent-child relation and the ordering of child nodes, and the semantic of each part of the structure, through labels assigned to tree nodes. The basic idea of TK function is the evaluation of the similarity between two tree structures by counting the number of substructures (i.e., *tree fragments*), that the trees have in common [95]. As TKs are a specific type of convolution kernel, various TK functions arise according to the particular kind of tree fragments they consider. In general, a TK function can be evaluated by adding, for each pair of nodes in both trees,

the contribution of all the common fragments in the trees rooted in these nodes. More formally, the similarity of two trees using a TK function can be evaluated as follows [95]:

Definition 4.1 *Given two tree structures T_1 and T_2, the similarity score assigned by a Tree Kernel to the trees is expressed as:*

$$K(T_1, T_2) = \sum_{n_1 \in T_1} \sum_{n_2 \in T_2} \Delta(n_1, n_2)$$

where n_1 and n_2 iterate on all nodes of the first tree T_1 and the second tree T_2, respectively. Δ is a non-negative function expressing the contribution of fragments in the sub-trees rooted in these nodes.

The type of considered tree fragments is delegated, in Definition 4.1, to the Δ function. Generally, the evaluated similarity value is a real positive number (i.e., in the range $[0, +\infty[$), regardless of the specific Δ function used to quantify the extent of fragments contribution, due to its non-negative constraint. However, in many practical cases a similarity score is more useful if it is expressed as a percentage. For this reason, it is possible to normalize a TK function [58, 31]:

Definition 4.2 *Given two ordered labelled trees T_1, T_2 and a TK function K, the* Normalized Tree Kernel *function K' derived by K is:*

$$K'(T_1, T_2) = \frac{K(T_1, T_2)}{\sqrt{K(T_1, T_1) \cdot K(T_2, T_2)}}$$

Given a tree structure, the maximum possible similarity value which a TK function could output, using that tree as one of its arguments, is the value obtained by evaluating the function on the tree with an equal one (both in structure and labels, and thus with that tree itself). For this reason, $K(T_1, T_1)$ and $K(T_2, T_2)$ are always greater or equal to $K(T_1, T_2)$ and thus, in Definition 4.2, the square root of the product between these self-similarities acts as a normalization factor. In fact, if the two trees T_1 and T_2 are equal, it follows that $K'(T_1, T_2) = 1$, representing an upper bound to the similarity. As a consequence, similarity scores produced by the normalized tree kernel belongs to the range $[0, 1]$. The normalized

TK function K' is indeed a kernel function: it is symmetric, as it can be easily proven that $K'(T_1, T_2) = K'(T_2, T_1)$, and positive defined, as the set of positive defined kernels is closed under product [58, 19]. Note that the normalization can be applied to any kernel function using the same operations, and not only to TKs.

Different TK functions have been proposed in literature, each of them characterized by a particular type of Δ function. These functions differ in the kind of fragments they consider during the evaluation of the contribution. For example, a Δ function could be designed to ignore labels assigned to nodes, in order to exclusively consider the structural relations in the evaluated trees. Among all the possible TK functions, some tree kernels have been extensively used in literature and are described hereafter.

To ease the mathematical formulation of the different types of Δ functions, the following notation will be used in the remainder of the section: if n is a node of a tree structure, its label will be referred as $l(n)$, while the sequence of its children will be denoted with $c(n)$ and the i-th child of n is denoted with the array-like notation $c(n)[i]$. Finally, the number of children of a node n will be denoted with $|c(n)|$.

4.1.3 The Sub-Tree Kernel

A simple type of tree fragments which can be considered in the evaluation of similarity between two tree structures is the *Sub-Tree*. Starting from a tree node n, the Sub-Tree rooted in n is the entire sub-structure including n and all of its descendant up to the leaves. Figure 4.2 presents some Sub-Trees rooted in different nodes of an ordered labelled tree (on the left). Note that a leaf node is the root of a Sub-Tree including only itself, e.g., the Sub-Tree with the only node labelled by f to the right of the figure.

The Sub-Tree Kernel (STK) function evaluates the similarity of two trees by counting the number of Sub-Tree fragments which the structures have in common. The traditional formulation of STK [31] ignores leaf nodes when evaluating the similarity. This is due to the fact that in many practical applications, leaf nodes are typically labelled with tokens and symbols which are unrelated to the structure of the tree. For example, syntax trees employed in the NLP domain assign the actual words of a

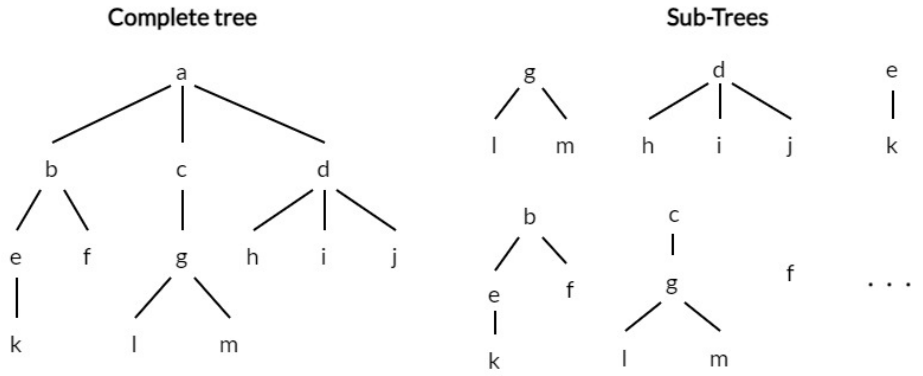


Figure 4.2. An example of some Sub-Tree fragments for an ordered labelled tree.

sentence to leaf nodes, while the information on the sentence's grammar structure is expressed totally by the inner tree nodes.

A definition of the Δ_{ST} function which characterizes the STK is the following [94]:

Definition 4.3 *Let n_1, n_2 be two nodes belonging to two different ordered labelled trees T_1, T_2, respectively. The Δ_{ST} function characterizing the STK can be evaluate according to the following cases:*

1. *if at least one among n_1 and n_2 is a leaf, or $l(n_1) \neq l(n_2)$, or $|c(n_1)| \neq |c(n_2)|$, then $\Delta_{\mathrm{ST}}(n_1, n_2) = 0$;*
2. *if $l(n_1) = l(n_2)$ and both child sequences of n_1 and n_2 are composed only by leaves, then $\Delta_{\mathrm{ST}}(n_1, n_2) = 1$;*
3. *otherwise, $\Delta_{\mathrm{ST}}(n_1, n_2) = \prod_{i=1}^{|c(n_1)|} \Delta_{\mathrm{ST}}(c(n_1)[i], c(n_2)[i])$*

As it can be seen by Definition 4.3, Δ_{ST} is formalized as a recursive function. Its possible values are only 1 or 0, and it can be seen as a Boolean function verifying whether or not n_1 and n_2 are roots of equal Sub-Trees. The first base step is applied when the nodes do not contribute to tree similarity. This includes the case of leaf nodes, which are ignored, the case in which the Sub-Tree fragments in n_1 and n_2 are surely different, when

nodes have different labels or different children. In this case, Δ_{ST} provides a null contribution. On the other hand, the second base step assesses a contribution equal to 1 to the two nodes when their labels match and there are only leaves beneath them. This means that the two nodes share the same semantic role in the tree structure and are parts of common Sub-Trees, ignoring leaves.

The third, recursive step applies when the two nodes have the same labels, the same number of children and any node in their child sequences is an inner node. In this case, Δ_{ST} provides a value of 1 only if the Sub-Trees rooted in the respective children are equal. To achieve this, Δ_{ST} is recursively evaluated on the pairs of children of n_1 and n_2 which share the same position in the child sequences. The Δ_{ST} values obtained for each pair are then multiplied together to obtain the contribution to the overall similarity. The product runs on all indexes of the child sequence $c(n_1)$, which are the same of those in $c(n_2)$ due to the condition $|c(n_1)| = |c(n_2)|$ needed to fall back into the recursive step. Note that any difference in the Sub-Trees rooted in n_1 and n_2 causes $\Delta_{\text{ST}}(n_1, n_2) = 0$, as in this case at least one Δ_{ST} value in the recursive step is 0 and therefore nullifies the product at all superior levels of recursion. Conversely, if the Sub-Tree fragments are equal, the contribution of nodes is 1. Hence, $deltast(n_1, n_2) \in \{0, 1\}$ for all tree nodes n_1, n_2.

A variation of STK, namely the STK+bow [95], also includes the contribution of leaf nodes according to a *bag-of-words* model [145]. In STK+bow, leaves contribute only to the overall similarity described in Definition 4.1, without affecting the Δ_{ST} function on inner nodes. To formalize the Δ function for STK+bow, it is possible to add another base step in Definition 4.3, before the first one:

0. $\Delta_{\text{ST}}(n_1, n_2) = 1$ *if* n_1 *and* n_2 *are both leaves and* $l(n_1) = l(n_2)$

By adding this new base step, if two leaves are labelled with the same value, their contribution will be equal to 1; otherwise, if they are not both leaves or their labels mismatch, the first base step applies, assessing a contribution of 0. This base case has the effect of adding the number of repetitions of each leaf node common to the two trees to the overall similarity evaluated by the STK+bow.

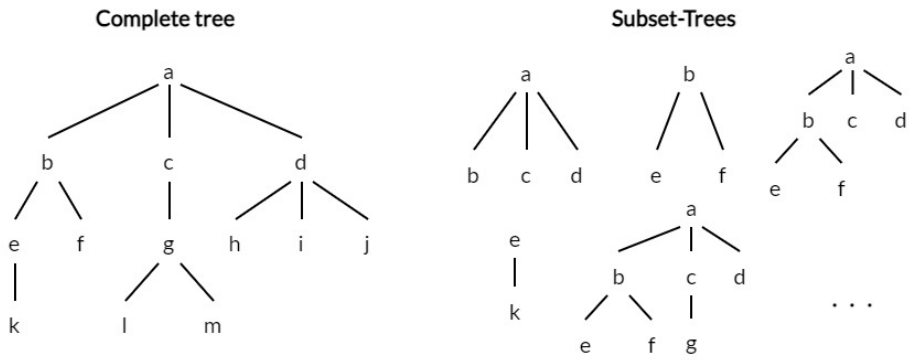


Figure 4.3. An example of some Subset-Tree fragments for an ordered labelled tree.

4.1.4 The Subset-Tree Kernel

Subset-Tree Kernel (SSTK) takes into account all common subsets between two trees to evaluate similarity. A *Subset-Tree* fragment, rooted in any node n, differs from a Sub-Tree fragment in the set of nodes included in the fragment: conversely from the latter, it is not mandatory for the former to include all descendants of n, and the included nodes can stop at any depth of the subtree. The only constraint in the set of node collection is that any sequence of child nodes should not be split, i.e., when a node is included in the fragment, all of its siblings should be inserted as well. Figure 4.3 shows an ordered tree and some of its Subset-Tree fragments. As it can be seen from the figure, it is not mandatory to include all nodes down to the leaves in a Subset-Tree, but the child sequences are either taken entirely, or not taken at all. As the collection of nodes can be interrupted at any depth, it is also possible to insert all descendant nodes, down to the leaves. Thus a Sub-Tree fragment is also a Subset-Tree fragment, causing the set of all Sub-Tree fragments to be strictly included in the set of all Subset-Trees. This allows the SSTK to consider a higher number of common fragments than STK when evaluating the similarity between the trees.

As seen with STK, the classic formulation of SSTK [31] ignores the contribution of leaf nodes as well, and for the same reasons. In this case,

the Δ function for SSTK can be expressed as follows [95]:

Definition 4.4 *Let n_1, n_2 be two internal nodes belonging to two different ordered labelled trees T_1, T_2, respectively. The Δ_{SST} function characterizing the SSTK is defined as:*

1. *if at least one among n_1 and n_2 is a leaf, $l(n_1) \neq l(n_2)$ or $|c(n_1)| \neq |c(n_2)|$ is true, then $\Delta_{\text{SST}}(n_1, n_2) = 0$;*
2. *if $l(n_1) = l(n_2)$ and both child sequences of n_1 and n_2 are composed only by leaves, then $\Delta_{\text{SST}}(n_1, n_2) = \mu$;*
3. *otherwise, $\Delta_{\text{SST}}(n_1, n_2) = \mu \cdot \prod_{i=1}^{|c(n_1)|} (1 + \Delta_{\text{SST}}(c(n_1)[i], c(n_2)[i]))$*

where $\mu \in]0, 1]$ is a decay factor penalizing the contribution of deeper Subset-Trees.

Definition 4.4 seems highly similar to STK in Definition 4.3, but possesses some crucial differences. Common points between the two definitions are the condition in which base and recursive steps are applied, and a contribution equal to 0 if no Subset-Tree fragment is common between the sub-structures rooted in nodes n_1 and n_2. On the other hand, to allow the evaluation of all Subset-Tree fragments of these nodes, the argument of the product in the recursive step has changed. In fact, the addition of 1 in the argument guarantees that the product will be at least 1 if two nodes have the same labels and the same number of children. If also the sequences of children match, the contribution of each pair of child nodes will be greater than 1. This can cause an exponential growth in the value of the product if the depth of the trees is large, as these terms are multiplied together. To mitigate this phenomenon, a *decay factor* μ is introduced both in the recursive step and in the second base step. This is a hyper-parameter of SSTK and can be tuned according to specific needs. In any case, the values produced by Δ_{SST} are not constrained in the range $\{0, 1\}$ as in the case of Δ_{ST}, but they are actually in the interval $[0, +\infty[$, scaling with the length of the child sequences and the depth of the sub-trees rooted in the considered nodes n_1, n_2.

To take into account the leaf nodes, a variant for SSTK has been also proposed, including the *bag-of-word model* for these terminal nodes,

namely SSTK+bow [95]. It is defined in the same manner as STK+bow, by adding the same preliminary base step, i.e., $\Delta_{\text{SST}}(n_1, n_2) = 1$ if n_1 and n_2 are leaves and are labelled with the same value. This variant of SSTK allows to add the number of common leaf nodes to the sum of contributions in Definition 4.1.

The recursive step of Δ_{ST} and Δ_{SST} for STK and SSTK, respectively, can be expressed with the same formulation, by using an additional parameter $\sigma \in \{0, 1\}$ which allows to distinguish between the two. Given two tree nodes n_1, n_2, this shared formulation [95] is:

$$\Delta_{\text{ST/SST}}(n_1, n_2) = \mu \cdot \prod_{i=1}^{|c(n_1)|} (\sigma + \Delta_{\text{ST/SST}}(c(n_1)[i], c(n_2)[i])) \tag{4.1}$$

When $\sigma = 0$ and $\mu = 1$, Equation 4.1 become precisely the recursive step in Definition 4.3, while if $\sigma = 1$, the recursive step of SSTK is derived.

Using Equation 4.1 it is possible to derive a dynamic programming algorithm [31] which evaluates STK and SSTK according to Definition 4.1 and using either Δ_{ST} and Δ_{SST} as Δ functions, as described in Algorithm 4.1. Aside from the two trees T_1, T_2, the inputs include also the decay factor μ and the value of σ to apply either STK and SSTK.

Both classical formulation of STK and SSTK ignore the contribution of terminal nodes, thus the algorithm initially prunes all leaf nodes from the input trees T_1 and T_2. Subsequently (lines 2-3), two sequences S_1, S_2 are initialized with the remaining nodes of T_1 and T_2 respectively, and ordered by increasing depth. This is due to the fact that the complexity of sub-problems in the recursive step is related to the depth of nodes.

For each pair of nodes in the two trees, the different values of Δ are stored in a $|T_1| \times |T_2|$ matrix. For any pair of nodes (n_1, n_2), the contribution $\Delta(n_1, n_2)$ will be stored in the corresponding cell of the matrix. The nested `for` loops in lines 5-20 evaluates whether the formulas in Definitions 4.3 or 4.4, by considering the nodes in the order they are present in the sequences S_1 and S_2. The first base step is implemented in lines 9-10, while the second is in lines 11-12. In particular, the `if` condition in line 11 checks if the nodes have the same labels and are parents of leaves only. As the leaves have been pruned, the latter check can be performed by controlling whether the length of any sequence of children is 0 (in the

algorithm, this check is applied in the child sequence of the first node).

Algorithm 4.1 Evaluation of STK and SSTK with a dynamic programming algorithm.
Input: T_1, T_2: two tree structures, $\mu \in]0,1]$: decay factor, $\sigma \in \{0,1\}$: switch between STK and SSTK
Output: K: the similarity score between T_1 and T_2

```
1:  Prune leaves from T1, T2
2:  S1 ← nodes in T1 ordered by ascending depth
3:  S2 ← nodes in T2 ordered by ascending depth
4:  Δ ← new |S1| × |S2| matrix
5:  for i = 1, ... |S1| do
6:     for j = 1, ... |S2| do
7:        n1 ← S1[i]
8:        n2 ← S2[j]
9:        if l(n1) ≠ l(n2) or |c(n1)| ≠ |c(n2)| then
10:          Δ[n1, n2] ← 0
11:       else if l(n1) = l(n2) and |c(n1)| = 0 then
12:          Δ[n1, n2] ← λ
13:       else
14:          Δ[n1, n2] ← λ
15:          for k = 1, ... |c(n1)| do
16:             Δ[n1, n2] ← Δ[n1, n2] · (σ + Δ[c(n1)[k], c(n2)[k]]
17:          end for
18:       end if
19:    end for
20: end for
21: K ← 0
22: for n1 ∈ S1 do
23:    for n2 ∈ S2 do
24:       K ← K + Δ[n1, n2]
25:    end for
26: end for
27: return K
```

The recursive step is applied in the `else` branch in lines 13-16, when

the nodes have equal labels and the same number of children. Initially, the Δ value is set to μ. Then, for each child in the sequences, this value is multiplied with σ plus the Δ of the children, previously stored in the matrix. After all the Δ contributions have been evaluated, the overall similarity K is evaluated by adding together all the values in the matrix (lines 21-26), and eventually returned.

The space complexity of Algorithm 4.1 is $O(|T_1| \cdot |T_2|)$, as it needs to keep a matrix of the same size to re-use Δ values. The time complexity is dominated by the nested `for` loops in lines 5-19. The two external loops iterate once for each pair of nodes in T_1 and T_1, for a total of $|T_1| \cdot |T_2|$ iterations. The `else` branch executes another loop on the sequence of children of considered nodes, to evaluate the recursive step. The number of iterations is equal to the length of the child sequence, and each iteration is executed in $O(1)$ time, retrieving the values stored in the Δ matrix. If the maximum number of any child sequence in the two trees, i.e. the *maximum branching factor*, is denoted with ρ, then the time complexity is $O(\rho \cdot |T_1| \cdot |T_2|)$.

In many practical contexts, there is no need to evaluate the entire Δ matrix explicitly, mainly because all pairs of nodes with different labels do not contribute to the overall similarity at all. Due to this observation, a *Fast Tree Kernel* approach has been proposed in literature [95, 131]. This approach is based on a preliminary extraction of all pairs of node which have the same labels, and then the main algorithm to evaluate the similarity is evaluated only on these pairs, as the contribution of other pairs is equal to 0. Although this implementation shows the same asymptotic time complexity, it is more efficient on average. In fact, the number of pairs of same-labelled nodes is often significantly less than the size of all pairs of nodes in the two trees. Moreover, as the set of different possible labels grows in size, finding two nodes with the same label is less likely.

4.1.5 The Partial-Tree Kernel

Partial-Tree Kernel (PTK) relaxes the constraint of the Subset-Tree Kernel, allowing to consider also sub-sequences of children of a given node. In this case, given a tree node n, a *Partial-Tree* fragment is a Sub-Tree which includes the node n, any of its descendant nodes at any depth, and any child can be included for each node. Some examples of partial trees

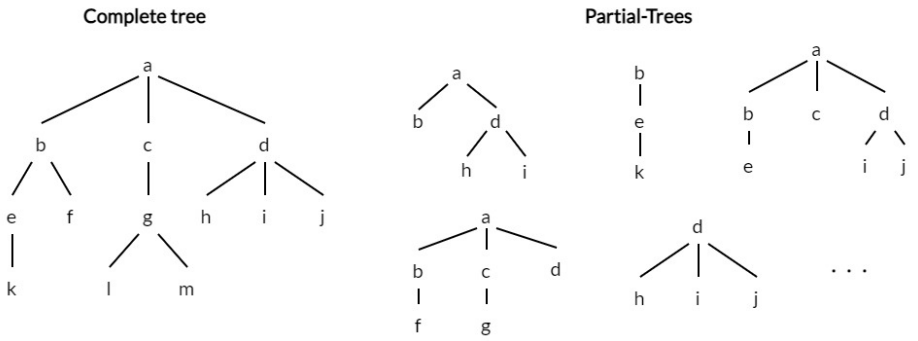


Figure 4.4. An example of some Partial-Tree fragments for an ordered labelled tree.

are shown in Figure 4.4. From the original tree (on the left), several partial trees can be extracted, picking nodes at any depth and with arbitrary subsequences of children, even those containing gaps between nodes. However, the *relative* position of nodes in the sequences should be preserved. This means that, given two nodes n, m in the same sequence of children with position i, j, which are included in a sequence of a Partial-Tree fragment with indexes k, l, if $i < j$ then must be $k < l$. As the definition of Partial-Tree allows to pick also the entire sequence of children, all Subset-Tree fragments are Partial-Tree fragments as well.

The PTK uses contributions of all Partial-Tree fragments in the two trees to evaluate their similarity. It is possible to derive a definition of the Δ_{PT} function employed in Definition 4.1 to formalize PTK, as follows [94]:

Definition 4.5 *Let n_1, n_2 be two nodes belonging respectively to trees T_1, T_2. The Δ_{PT} function defining PTK is:*

1. *if $l(n_1) \neq l(n_2)$, then $\Delta_{\text{PT}}(n_1, n_2) = 0$;*

2. *if $l(n_1) = l(n_2)$ and n_1, n_2 are both leaves, then $\Delta_{\text{PT}}(n_1, n_2) = \tau$;*

3. otherwise,

$$\Delta_{\mathrm{PT}}(n_1, n_2) = \mu \cdot \left(\lambda^2 + \sum_{\substack{J_1, J_2: \\ |J_1|=|J_2|}} \lambda^{d(J_1)+d(J_2)} \cdot \prod_{i=1}^{|J_1|} \Delta_{\mathrm{PT}}(c(n_1)[J_{1,i}], c(n_2)[J_{2,i}]) \right)$$

where $\tau \in [0,1], \mu, \lambda \in]0,1]$ are the terminal factor, the decay factor and the gap-penalization factor, respectively. $J_1 = (J_{1,1}, \ldots, J_{1,|c(n_1)|})$ and $J_2 = (J_{2,1}, \ldots, J_{2,|c(n_2)|})$ are sequences of positions in children of n_1 and n_2, respectively.

Differently from the STK and SSTK functions, the PTK can take into account also the leaves of the trees. More in details, the weight of the contribution of leaves can be set through an additional parameter, the *terminal factor* τ. This parameter is in the range $[0, 1]$ and determines how much the overall similarity is conditioned by leaves with common labels. If $\tau = 0$, common leaves are ignored, while when $\tau = 1$ their contribution is fully included as in the STK+bow and SSTK+bow. Any intermediate value allows to set different weights to leaf nodes and can be tuned according to the specific application. Furthermore, conditions of the Δ_{PT} base steps do not involve the child sequences of the nodes. Indeed, when considering Partial-Tree fragments, all the sub-sequences of children should be compared. Even if the full sequences are not equal, some of the sub-sequences could, and thus they contribute to the evaluation of Δ_{PT}.

The recursive step for Δ_{PT} presented in Definition 4.5 deeply differs from the previous Δ_{ST} and Δ_{SST} formulations. Although it presents the same decay factor μ to prevent the exponential growth of the output, it also adds together all the contributions for sub-sequences. In the above notation, a sub-sequence is represented with a sequence of indexes, each of which refers to a specific position in the child nodes. As the Partial-Trees fragments should match entirely to contribute to the similarity, the sequence of indexes J_1, J_2 which Δ_{PT} evaluates are those with the same length (this is expressed by the constraint $|J_1| = |J_2|$ in the summation). Note that J_1, J_2 can be of any length in $1, \ldots \min\{|c(n_1)|, |c(n_2)|\}$. The Δ_{PT} function is therefore evaluated on all child nodes in the position ex-

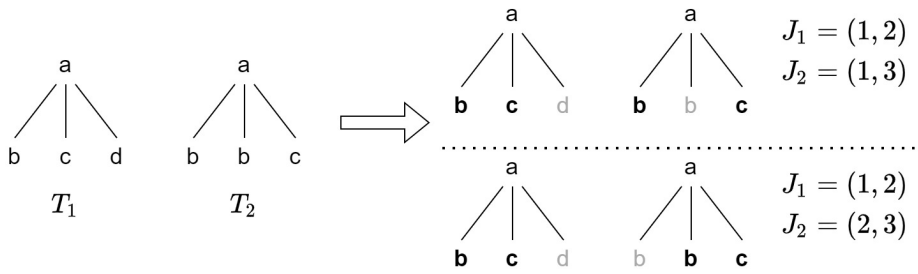


Figure 4.5. An example of gaps in Partial-Tree fragments. On the left, the original trees. On the right, two matching sub-sequences, either with and without gaps.

pressed by the respective indexes in J_1 and J_2, and all the results multiplied together. If any Δ_{PT} of the considered children is 0, then the two Partial-Trees are different, and do not contributes to the similarity.

Partial-Tree fragments include also sub-structures with gaps in the sequence of children. Although these fragments can possibly contribute to the similarity, the matching child nodes can be very scattered in the sub-sequences and, in some case, they should be penalized. As an example, consider Figure 4.5. When evaluating the two ordered labelled trees, on the left, PTK should consider the two sub-sequences of nodes labelled with "a" depicted on the right. Matching nodes in the sub-sequences are denoted in bold. In the topmost matching sequences, with indexes $J_1 = (1, 2)$ and $J_2 = (1, 3)$, the "b" node, belonging to T_1 and in first position, is matched with its homologous in the second tree; the subsequent "c" node in T_1 is however matched with node "c" in third position, introducing a gap in the matched sub-sequences. On the other hand, the bottom sub-sequences involve the sub-sequences with indexes $J_1 = (1, 2)$ and $J_2 = (2, 3)$. In this case, the "b" node in first position is matched with the same node in second position, and the subsequent "c" node is matched with the other "c" node in third position. In this latter case, no gap is present in the sequences, and the involved sub-structures could have a more similar topology.

Specific applications can treat gaps in Partial-Tree fragments differently. To allow a higher rate of flexibility, PTK provides a gap penalization factor λ weighting the impact of gaps in sub-sequences. This parameter is tunable according to the particular application needs. If $\lambda = 1$,

there is no penalization for gaps, and all sub-sequences are treated equally. Conversely, the more λ approaches 0, the less is the contribution of sub-sequences possessing gaps with respect to the original sequences. To effectively evaluate the penalization factor for the sub-sequences, λ is raised to the power of the number of gaps therein. To measure the number of gaps, Δ_{PT} uses a function $d(J)$, which maps an index sequence J to the number of gaps. The d function can be easily evaluated as $d(J) = J_{|J|} - J_1$, i.e., the difference between the last and the first index in the sub-sequence of indexes.

A naïve algorithm based on the Δ_{PT} in Definition 4.5 leads requires exponential time. However, it is possible to apply dynamic programming to evaluate similarity in polynomial time [94]. To do so, let Δ_p be a function evaluating the contribution of common sub-sequences of children composed by exactly p nodes. Using Δ_p, it is possible to rewrite Δ_{PT} in Definition 4.5 as:

$$\Delta_{\text{PT}}(n_1, n_2) = \mu \cdot \left(\sum_{p=1}^{\min\{|c(n_1)|, |c(n_2)|\}} \Delta_p(c(n_1), c(n_2)) \right) \tag{4.2}$$

This alternative and equivalent formulation sums the contribution of all Partial-Tree fragments according to the length of their sub-sequences. All sub-sequences are included in the evaluation of Equation 4.2, from length 1 to the maximum possible length, which is precisely the size of the shorter sequence of children between n_1 and n_2.

To derive an explicit formulation of Δ_p, it is possible to rewrite the sequence of children of a tree node n as $c(n) = s \cdot m$, where m is the last child, and it is concatenated with the sequence of remaining children s. Thus, given two nodes n_1 and n_2, their sequence of children can be rewritten as $c(n_1) = s_1 \cdot a$ and $c(n_2) = s_2 \cdot b$, where a, b are the last nodes and s_1, s_2 the remainder sequences. Denoting with $s[i : j]$ the sub-sequence of s starting from the i-th node up to the j-th node, the value of Δ_p can be recursively evaluated as follows:

$$\Delta_p(s_1 \cdot a, s_2 \cdot b) = \Delta_{\text{PT}}(a, b) \cdot \sum_{i=1}^{|s_1|} \sum_{j=1}^{|s_2|} \lambda^{|s_1|-i+|s_2|-j} \cdot \Delta_{p-1}(s_1[1 : i], s_2[1 : j]) \tag{4.3}$$

The above formulation of Δ_p is recursively computed through the contribution of shorter sequences Δ_{p-1}, as if the last nodes a and b have the same labels (thus, $\Delta_{\text{PT}}(a,b) \neq 0$), they contribute once to each sub-sequence of length $p-1$ that can be created from s_1 and s_2. The algebraic manipulation of Equation 4.3 allows to derive a more useful recursive expression. Let D_p be the double summation in Equation 4.3, it follows that:

$$D_p(k,l) = \Delta_{p-1}(s_1[1:k], s_2[1:l]) + \lambda \cdot D_p(k, l-1) + \lambda \cdot D_p(k-1, l) + \lambda^2 \cdot D_p(k-1, l-1) \tag{4.4}$$

This formulation of D_p can be derived by extracting the longest sequences from the double summation. The first additive term in Equation 4.4 refers to the actual value of Δ_{p-1} on the longest sequences. The second and third terms include the summation of all contributions when one of the sequence is reduced by one element, and they are both multiplied by λ as an effect of reducing the sequence length. Similarly, the last term refers to the summation of all contributions when the last elements of both sequences are not considered, and it is multiplied by λ^2 due to the reduction in both lengths.

Using the D_p function, Δ_p in Equation 4.3 can be expressed as:

$$\Delta_p(s_1 \cdot a, s_2 \cdot b) = \begin{cases} 0 & \text{if } l(a) \neq l(b) \\ \Delta_{\text{PT}}(a,b) \cdot D_p(|s_1|, |s_2|) & \textit{otherwise} \end{cases} \tag{4.5}$$

Combining Equation 4.5 and Equation 4.2 together, it is possible to derive a dynamic programming algorithm to evaluate Δ_{PT}. Given two nodes n_1 and n_2, Δ_{PT} can be evaluated by using a $p \times |c(n_1)| \times |c(n_2)|$ matrix, to temporarily store the solutions to every sub-problem, for each sequence length and each sub-sequence of children. This results in a time complexity of $O(p \cdot |c(n_1)| \cdot |c(n_2)|)$, needed to fill the matrix. Furthermore, the evaluation of Δ_{PT} should be evaluated for all pairs of nodes of the trees T_1, T_2 in evaluation, as according to Definition 4.1 the contributions should be added together. Denoting with ρ the maximum branching factor between T_1, T_2, the total complexity of the algorithm is $O(\rho^3 \cdot |T_1| \cdot |T_2|)$. This means that the similarity of wider trees is more expensive to evaluate through PTK.

4.2 Tree Kernels for Source Code

This section describes how TK functions can be applied to evaluate the extent of changes in the source code, through its AST representation. Initially, the section presents a description of ASTs, highlighting their flexibility in comparison with the more rigid *parse trees*. Subsequently, the section explains how it is possible to employ the similarity measure of TKs to quantify differences in the source code, and presents a qualitative discussion on which function could be more suitable to assess the changes in ASTs.

4.2.1 Abstract Syntax Tree Representation of Source Code

Developers see the source code as a sequence of characters, writing and modifying it according to rules specified by the particular programming language they use. This textual representation is easily understandable by humans, but has the flaw of being hard to automatically analyze: the extraction of structural relations between the different units of the code, such as statements in blocks, or expressions in statements, cannot be straightforwardly processed using this representation. Complex structural relations between fragments of source code are typical of all modern high-level programming languages, and these relation can be defined by tree-based models. Tree-based models naturally arise from source code and are employed in several tasks where a refined analysis of the source is needed [99, 14].

The basic tree-based representation of source code is through *Parse Trees*. Parse Trees are a direct transformation of a programming language grammar rules to an ordered labelled tree-based model, including language-specific symbols such as semicolons, brackets, or keywords [2]. Each *production rule* in the grammar is transformed in a node, and labelled with the role of the applied rule. The relations between nodes are made explicit by parent-child relations between nodes, according to the production rules involved in the construct. The application of nested rules is expressed as child sequences, and the position of a child in the sequence relates to the position in which the rule is applied in the source code. However, this kind of modelling source code is a mere tree-structured representation of grammar rules, and the produced trees are highly correlated to the spe-

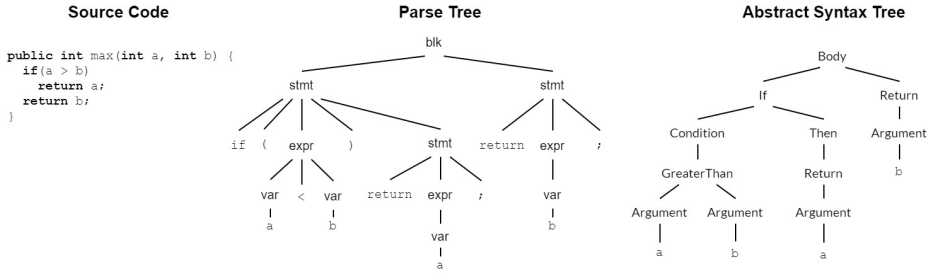


Figure 4.6. Comparison between a Parse Tree and an AST representation for a simple Java method returning the maximum between two integers.

cific language. For this reason, the specific structure of parse trees can be significantly different between different programming languages. Furthermore, Parse Trees present a high-level of redundancy among their nodes. In fact, the keywords of the language are included in the model, usually as terminal nodes. This causes Parse Trees to grow both in depth and in width, although these nodes do not add any further information to the structure.

On the other hand, ASTs are an ordered labelled tree-based representation of source code, not as tightly coupled to language grammar rules as Parse Trees are. Indeed, their structure is more flexible, allowing both the pruning of redundant nodes and the re-arrangement of the tree topology, to highlight specific aspects for the problem at hand. Moreover, the set of labels is not fixed *a-priori*, allowing to use more meaningful labels to mark the contained nodes. Labels can in fact be freely chosen according to specific tasks, even if they are typically related to the various constructs of the programming language. Leaves model the *tokens* in the source code, such as the name of a variable or a literal value. Differently from Parse Trees, ASTs seldom include language keywords in their structure, as this information can be usually derived by the semantic role of inner nodes.

Figure 4.6 shows a comparison between the Parse Tree and the *AST* representation of a simple Java method. The Parse Tree (in the middle) clearly represents the structure of the construct of the source code through parent-child relations, and the order of child nodes mimics the appearance of constructs in the code. However, it presents also nodes for keywords (terminal node labelled with `if`) and symbols as `(`, `)` and `;`. These lat-

ter nodes, however, have no effect on the semantics of the sub-tree, and can be highlighted from the surrounding structure. Furthermore, different programming languages can use different symbols to separate statements and to specify constructs, and the Parse Tree of an equivalent method can significantly differ. For example, as Python does not use any symbol to end a statement[2] or does not need brackets in branch and loop constructs, a method implementing the same functionality would not have any `;`, `(` or `)` characters, and then its Parse Tree would significantly differ. Furthermore, labels of inner nodes provide little information of the semantic role of constructs. For example, the leftmost `stmt` node is not immediately recognizable as the node representing the branch construct `if` in the source code. To deduce its role, it is necessary to navigate its sub-tree to find the `if` terminal node. The same happens for the `expr` node in the statement, which semantically represents the branching condition, and the particular semantic of the condition, in which the operator `<` is modelled as a terminal node in the expression. Due to these phenomena, it is typically not straightforward to understand the semantics of all nodes.

The AST for the same method in Figure 4.6 (on the right) is instead more concise and informative. As it can be seen, the *Body* node modelling the body block of the method has two children, the first one representing the *If* block, and the second one modelling the *Return* statement. The *If* block, in turn, has two children, a *Condition* node holding the check performed by the `if` statement, and a *then* sub-tree which includes the *Return* statement in the branch. The *Condition* sub-tree includes the greater expression, represented as a node labelled with *GreaterThan*, and the values on which it is performed (the two *Arguments*. Variable names are leaf nodes, carrying the information of specific tokens included in the expressions. As in Parse Trees, the position of children in sequences mimic the order in which statements or expressions appear in the source code also in ASTs. In the example, as the `if` statement occurs before the last `return` statement, the *If* sub-tree is placed before the upmost *Return* sub-tree; furthermore, the left operand of the greater condition, represented by the leftmost *Argument* node related to the variable `a`, is placed before the node modelling *Argument* `b`. This positional ordering allows to keep

[2]More precisely, an *end-of-line* character is used to end a statement in almost all circumstances.

information on the order of statements and operands in the original code. Conversely from Parse Trees, specific symbols used in the language have been pruned, to keep the representation more concise. Furthermore, the semantic role of nodes in the AST can be directly identified just analyzing labels, without the need of navigating the sub-trees as it happens in Parse Trees.

Generally speaking, AST models provide a more concise representation of source code structure, both increasing the meaningfulness of the representation and pruning nodes related to language-specific symbols and keywords. Since AST representations are not defined by the mere application of grammar rules, it is possible to employ different AST representations of source code, which are specifically tailored towards particular tasks. This flexibility guarantees a higher level of transparency with respect to programming languages with respect to Parse Trees, and thus ASTs can be designed to be language-agnostic and more generally applicable as underlying models for many applications. For these reasons, in order to evaluate the structural similarity between source code, the AST representation of source code has been chosen as the underlying model on which the similarity, using TKs, is evaluated.

4.2.2 Measuring Changes in Source Code with Tree Kernels

The meaningful representation of source code through ASTs can be readily evaluated by TK functions, as the representation is based on ordered labelled trees. A straightforward application of TK on two ASTs can produce a measure of similarity between two source code elements, under the type of the particular Tree Kernel and normalization employed. Granularity of code elements can be chosen according to specific needs, ranging from AST modelling an entire module composed by a collection of source files, to finer-grained AST representing single statements. Usually, a good trade-off between these two levels is represented by file and function granularity levels (respectively, class and method granularity in object oriented programming languages).

The evaluation of similarity can be useful in many tasks of code-analysis. However, churn-based RTP techniques need a measure of the extent of code changes, rather than similarity. To this purpose, it is pos-

sible to use TK functions to derive how different two tree structures are, i.e., their *dissimilarity*. The formulation of TKs cannot be directly used to measure dissimilarity, as they generally provide a similarity value in the range $[0, +\infty[$ and thus a quantification of the differences between trees is not straightforward. However, a normalized kernel function produces similarity values in $[0, 1]$ due the normalization factor, and thus a reasonable extent of the dissimilarity can be obtained by simply complementing this similarity. More formally:

Definition 4.6 *Given two ordered labelled trees T_1, T_2 and a TK function K, their dissimilarity can be expressed as*

$$K_{\text{diss}}(T_1, T_2) = 1 - K'(T_1, T_2)$$

where K' represents the normalized kernel function in Definition 4.2.

This measure allows to quantify the percentage of changes between the two tree structures. If the two trees are equal, $K_{\text{diss}}(T_1, T_2) = 0$ as $K'(T_1, T_2) = 1$; conversely, the more K_{diss} approaches to 1, the more the percentage of normalized kernel similarity gets close to 0.

Another manner to quantify the amount of changes in source code, represented by ASTs, is to define a *distance* measure between the tree structures. A distance measure can be derived using TK functions, i.e., the *Tree Kernel Distance*:

Definition 4.7 *Given two ordered labelled trees T_1, T_2 and a TK function K, the* Tree Kernel Distance *d_{K} between T_1 and T_2 is*

$$d_{\text{K}}(T_1, T_2) = K(T_1, T_1) + K(T_2, T_2) - 2 \cdot K(T_1, T_2)$$

The formulation of *Tree Kernel Distance* d_{K} derives from the more general definition of *Kernel Distance* [119], which can be applied to any kind of kernel function. The distance is evaluated by summing the *auto-similarity* of the trees with themselves, and then subtracting the similarity evaluated by the kernel between the trees. The function d_{K} is actually a distance in the high-dimensional space implicitly defined by a kernel function. In fact, for all trees T, the distance of T with itself is 0, i.e., $d_{\text{K}}(T, T) = 0$, as it clearly follows from Definition 4.7. It is always positive,

as for all trees T_1, T_2, the maximum possible similarity is given by the kernel on each tree with itself, thus in any case $K(T_1, T_1) \geq K(T_1, T_2)$ and $K(T_2, T_2) \geq K(T_1, T_2)$. By summing these inequalities together and rearranging, it follows that $d_{\mathrm{K}}(T_1, T_2) \geq 0$. Furthermore, it is symmetric, as $d_{\mathrm{K}}(T_1, T_2) = d_{\mathrm{K}}(T_2, T_1)$, due the symmetry of kernel K. d_{K} satisfies also the *triangle inequality*, i.e., for all triples of trees T_1, T_2, T_3, $d_{\mathrm{K}}(T_1, T_2) \leq d_{\mathrm{K}}(T_1, T_3) + d_{\mathrm{K}}(T_2, T_3)$.

Among the different types of TK functions defined in literature, the PTK has been successfully applied in several code related tasks, due to the fact that it is able to consider Partial Tree fragments. The main motivation is that PTK is more sensitive to common operations which are performed to the source code during evolutionary steps. In fact, typical changes in the code involve the addition or the removal of expressions, statements, or blocks, and those operations are reflected in the addition or removal of nodes in the AST representation. The ability of PTK to take into account also Partial-Trees allows it to produce a similarity value which is more sensitive to these kinds of operations than STK and SSTK. Furthermore, there are software evolutionary practices in which a high number of changes reflects small extents of semantical modification, such as refactoring (e.g., when the changes are limited to the simple renaming of a variable).

As an example, consider the modification of the `foo` method in Figure 4.7. The `foo_refactored` method shows a renaming of the local variable `x` to `n`. A naïve textual approach would mark every line as changed, as all lines in the refactored version are different with respect to the original method. As it can be seen by the AST representation, renaming only affects labels of leaf nodes related to the former `x` variable, while the tree structure remains untouched. An application of STK to the original and refactored method, however, assesses a rather low normalized similarity score of 38%. This is due to the fact that almost every sub-tree fragment is changed between the two methods, therefore the result of summation in Definition (4.1) is small. On the contrary, PTK produces a similarity score of 93% due to the higher number of partial-tree fragments which match between the original and the refactored method.

When the `foo` method is changed by substituting its last line and semantically performing different operations, a textual difference will mark

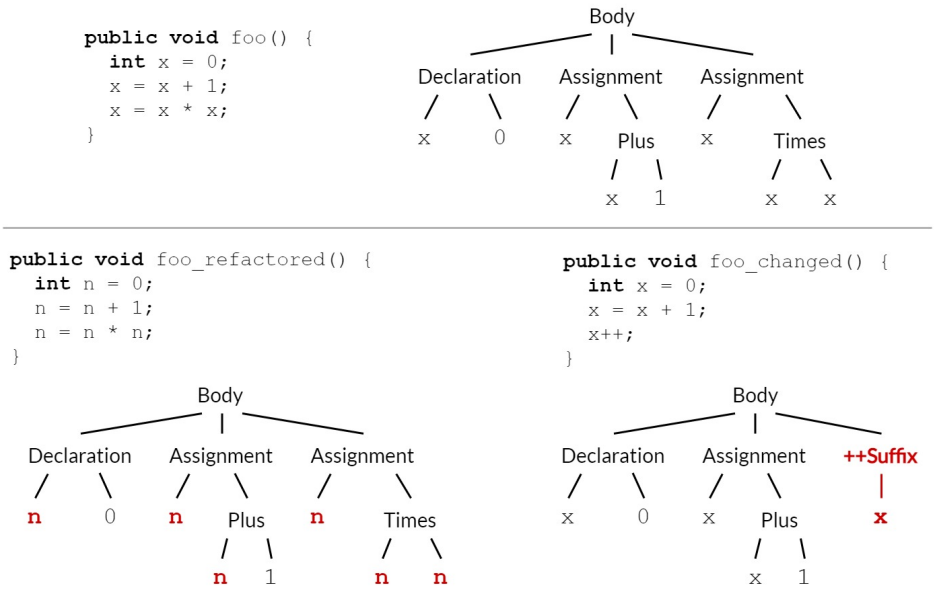


Figure 4.7. Different changes in AST representations according to changes in the source code. The original `foo` method (left) is refactored by renaming its variable (center) and with both the removal of a statement and the addition of a new one (right). Changed nodes are highlighted.

just this line as changed. The changes in the method's AST are however more pronounced, as a whole sub-tree prominently differs, while the others remain unmodified. In this case, the STK returns a normalized similarity score of 91%, as just one sub-tree fragment does not contribute to the total. Instead, PTK outputs a score of 73% due to the higher number of mismatching fragments, thus the evaluated similarity results lower. As shown by the example, considering partial trees can produce a more refined score both with respect to a naïve textual similarity, and to TKs considering different kinds of fragments.

Concerning the tuning of PTK hyper-parameters, as described in Definition 4.5, an empirical investigation on their impact on different real-world code changes has been carried out, using a *grid search*. None of the tried values provided the best results on all considered changes, but on average a decay factor $\mu = 0.4$, a gap penalization factor $\lambda = 0.7$ and a termi-

nal factor $\tau = 0.5$ generally resulted in a more accurate evaluation of the structural similarity.

4.3 Prioritizing Test Cases with Tree-Kernels

During the evolution of software due to maintenance activities involved in its life cycle, developers update the source code of different modules and packages, in order to correct unintended behaviors, add new functionalities or optimize previously implemented requirements. These activities typically lead to performing changes in the source code implementing already validated modules, and the very structure of source code could be deeply modified. These changes in the source code structure can easily be prone to mistakes, therefore invalidating some of the software requirements. For these reasons, a fine-grained analysis of the code churn could be employed in order to reduce the efforts of regression testing, and to drive the process of re-validation on code sections which have been subjected to a greater extent of changes.

This section presents MTK, a novel RTP approach which employs the PTK to exploit and quantify the structural differences between two versions of a software, ranking test cases based on the extent of modified functions/methods they cover, with the goal of producing a permutation. Moreover, the section defines an additional approach, namely MTK-QS, designed to enhance the rate of coverage of changed methods, to accelerate the exercising of modified parts in the code. MTK-QS embed a more general Quotient-Set Prioritization (QS) approach, which can be generally applied as a complementary step to any prioritization strategy which rank test cases by the means of some evaluated priority scores. Both MTK and MTK-QS have been proposed in [8]. Without loss of generality, the description of the techniques is based on the fact that the software to which they are applied follows the object oriented paradigm, but can be easily extended also to languages using different paradigms.

4.3.1 Tree-Kernel Based Prioritization

The novel MTK strategy leverages *code churn* information along with past *coverage reports* to prioritize test cases in a test suite. This strategy

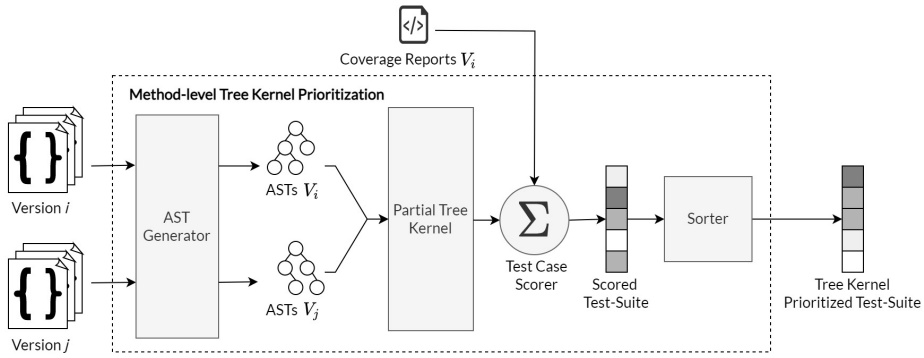


Figure 4.8. Execution steps for the MTK prioritization approach.

employs PTK to quantify the extent of changes between two versions of a software project, taking into account not only their number, but also their impact on the syntactic structure of the program. Specifically, MTK employs an evaluation of changes using the dissimilarity measure K_{diss}, in Definition 4.6, with K the PTK function.

An overview of all steps to evaluate the MTK strategy is presented in Figure 4.8. As any RTP approach, its application context is during the evolution of the software from a *previous* version V_i to a *current* version V_j. MTK receives in input the source code of both software versions V_i and V_j, along with the test coverage reports of the previous version. Coverage reports should be at least at method-level or finer granularity. An *AST Generator* sub-module is responsible to extract the ASTs of all methods in the two versions, which will be then fed to the PTK to evaluate the extent of their changes. A *Test Case Scorer* module joins the coverage reports with code-churn information, in order to assign a priority score to all test cases. Eventually, the test cases are re-arranged in descending order of scores, and the prioritized test suite returned.

To extract the ASTs from source code, initially the Parse Trees of code units are produced through a grammar parser based on the rules of employed programming language. Once Parse Trees have been obtained, the generator module models them in a higher level of abstraction, to produce the ASTs used by the subsequent component. As the granularity is considered at method-level, each constructed AST refers to one specific

method in the source code of a software version. However, to compare the evolution of a particular method from the previous to the current version, is necessary to link the AST of methods in the previous versions to those in the current version. To this end, a *method matching* $\mathscr{M}$ should be defined between the two versions. Given a previous software versions V_i and a current software version V_j, a method matching $\mathscr{M}$ between the versions is the set of pairs (m_p, m_c) of methods whose source code has been changed between V_i and V_j. m_p represents the method in the previous version V_i, while m_c is the method in the current version V_j under test.

The construction of the method matching $\mathscr{M}$ relies on *qualified names* of methods to identify the pairs. More specifically, $m_p \in V_i$ and $m_c \in V_j$ are matched if they have the same signature and are declared in packages and classes sharing the same name. If no method satisfies this constraint, that method remains unmatched. As the qualified name of the method is generally unique across all the source code, every method is matched at most once. Note that, using this approach, methods that have been renamed between versions V_i and V_j are not matched. This choice has a two-fold rationale. On one hand, if the method has been simply renamed between the two versions, with no changes to its source code, the dissimilarity obtained with the TK would be 0, so the method will not contribute to the score of any test cases covering it. On the other hand, in cases where there are also changes to the source code, it is not straightforward to discriminate between a freshly-added method and a method that was both renamed and significantly modified. Doing so would require setting some similarity threshold to determine whether a method is an evolved version of a previously existing method, or a new method entirely. Defining such a heuristic approach for matching looked arbitrary and not straightforward, thus no approach has been involved to this purpose.

The pseudo-code for the evaluation of the method matching $\mathscr{M}$ is presented in Algorithm 4.2. The algorithm creates a map with all methods in the previous version V_i (lines 1-4). The key of the map is the *qualified signature* of methods, i.e., a string obtained by concatenating the full classpath of the method, including the package, classes, method name and the type of all of its arguments. The name of the formal arguments of a method is not included, as two functions or methods with the same argument types and positions are indiscernible from each other in several programming

Algorithm 4.2 Construction of the matching between methods of two software versions.
Input: V_i: previous version sources; V_j: current version sources
Output: $\mathscr{M}$: the method matching, containing the pairs for methods whose qualified names match between the two versions.

```
1: methods_p = []
2: for each m_p ∈ V_i do
3:    methods_p[m_p.qualifiedSignature]= m_p
4: end for
5: M = ∅
6: for each m_c ∈ V_j do
7:    if methods_p.hasKey(m_c.qualifiedSignature) then
8:       m_p = methods_p[m_c.qualifiedSignature]
9:       M = M ∪ (m_p, m_c)
10:   end if
11: end for
12: return M
```

languages (such as Java, C and C++). Then, for each method m_c in the current version V_j, the algorithm checks if the qualified signature of m_c is present in the map of the previous method. If the check is successful, the previous method m_p is retrieved from the map and the pair (m_p, m_c) is added to the matching $\mathscr{M}$, on lines 7-9. Eventually, the method matching $\mathscr{M}$ is returned in line 12.

MTK then evaluates the churn information for all paired methods. In particular, for a pair of matched method $(m, m') \in \mathscr{M}$, the amount of changes between them is evaluated trough the PTK, applying the kernel dissimilarity presented in Definition 4.6. In particular, if $s(m, m')$ is the similarity score of m and m' ASTs evaluated by the normalized PTK, the dissimilarity score is $1 - s(m, m')$. Note that in case of absence of changes in the body of a method, the dissimilarity is 0 due to the fact that the respective ASTs are equal. Thus, if there are no modifications in a method between the two versions, the evaluation of PTK is redundant as it is possible to verify this condition through its textual representation: if the source code of the method has not been modified (excluding comments), then a dissimilarity score of 0 can be directly assessed to the method. This

observation can speed up the evaluation of MTK: if a preliminary equality check of the respective source code returns true, then their dissimilarity is directly set to 0; otherwise, the normalized PTK is applied normally. As a textual comparison is generally computationally cheaper than the application of PTK, this heuristic can significantly speed up its evaluation with software which have a high number of methods, but which underwent to a small number of changes between previous and current versions.

To join test cases and the churn information obtained via PTK dissimilarity, MTK leverages test coverage reports of the previous version V_i. For prioritization purposes, MTK considers MTK only those test cases which are present in both versions of the software. In fact, test cases which have been removed in the current version cannot be executed anymore, thus are excluded from prioritization. On the other hand, test cases which have been added in the current version lack of coverage information, as their execution has not been performed yet. Usually, new test cases have been introduced to test freshly added functionalities in the current software version, or to enforce the validation of previous software components which were not adequately tested. Several policies can be applied in RTP scenarios when new test cases are present (e.g., execute the new test cases before preexisting ones), but usually these cases are not considered in regression testing studies, as their number and their execution cost are typically negligible with respect to those already present in the entire suite.

The priority score assessed to a test case by MTK is the sum of all the amounts of changes of methods covered by that test. More formally, the score function for a test case can be defined as follows:

Definition 4.8 *Let T be a test cases, and* $\mathrm{score}(T)$ *be the set of all methods covered by T in the previous version, and let $\mathscr{M}$ be the matching of methods between previous and current versions. The* score *of test T is:*

$$\mathrm{score}(T) = \sum_{\substack{m_p \in Cov(T) \\ \exists m_c : (m_p, m_c) \in \mathscr{M}}} (1 - K(m_p, m_c))$$

where K denotes the normalized PTK function.

The summation in Definition 4.8 applies to all methods in the previous version which are covered by test case T and which have been paired with

Algorithm 4.3 Algorithm to evaluate MTK Prioritization.
Input: TS: test suite for current version, V_i: previous version sources, V_j: current version sources, score$_i$: method-level coverage information for each test case in version V_i
Output: TS': reordered test suite

```
1: 𝓜 ← createMethodMatching(V_i, V_j)
2: for each (m_p, m_c) ∈ 𝓜 do
3:    if m_p.text ≠ m_c.text then
4:       K[m_p, m_c] ← normalizedPTK(m_p.ast, m_c.ast)
5:    else
6:       K[m_p, m_c] ← 1
7:    end if
8: end for
9: for each T ∈ TS do
10:   T.score = 0
11:   for each m_p ∈ score(T) do
12:      if ∃m_c : (m_p, m_c) ∈ 𝓜 then
13:         T.score = T.score +(1 − K[m_p, m_c])
14:      end if
15:   end for
16: end for
17: TS' ← sortByScoreDescending(TS)
18: return TS'
```

a method in the current method in the matching $\mathscr{M}$. After the score function has been evaluated for all test cases, the test suite is eventually sorted by these scores in descending order.

The pseudo-code for MTK is presented in Algorithm 4.3. The first line of the algorithm calls the auxiliary function *createMethodMatching*, which executes the steps in Algorithm 4.2 to create the set of matching methods $\mathscr{M}$. Lines 2-8 evaluate the similarity between the pairs of methods in $\mathscr{M}$, and save these values in the K dictionary, whose keys are the pairs of methods. Initially, the algorithm checks whether the textual representations of the method bodies are different (line 3). If so, similarity between the methods ASTs is evaluated using the auxiliary function *normalizedPTK*. This function evaluates the similarity through PTK, implemented using

the formulation in Definition 4.1 with Equations 4.2 and 4.5, and then normalizes the resulting value according to Definition 4.2. If the text of the two methods is equal, the similarity is directly set to 1. Lines 9-16 evaluate the score for each test case in the suite. For a test case, the algorithm considers all the methods covered by the test in the previous version, checking if a method is paired in the current version through the matching $\mathscr{M}$. If such a pair exists, the dissimilarity score of the pair is added to the test score. The test suite is then sorted by the evaluated scores in descending order and returned (lines 17-18).

4.3.2 Quotient-Set Prioritization

As it can be noted from Definition (4.8), the score function assigns identical values to test cases covering the same set of changed methods. Conversely, test cases with the same scores can cover different changed methods, given that the structural changes in the covered methods are equal. However, this scenario is unlikely in practice, and implies that two test cases cover different methods subjected to very identical changes. Therefore, test cases with the same score have high possibilities to cover the exact set of changed methods.

The MTK method defined in Section 4.3.1 orders the test suite according to score function values. This means that all the test cases having the same score are scheduled one after another in the produced permutation. This can lead to the execution of several test cases which repeatedly stress the very same sections of changed code, degrading the rate of fault-detection. In fact, faults introduced in different changed methods will be stressed by other test cases, which have unfortunately been postponed in the ordering due to a lower assigned score. The main reason behind this behaviour is that the score function is *local* to a single test case, not considering which parts of the code might have been already exercised by test cases earlier scheduled. In the worst-case scenario, this behaviour leads to many undetected faults when the test suite execution is interrupted due to expiration of resources allocated for the testing phase.

For this reason, an evolution of the MTK approach has been designed, to better diversify the coverage of changes for the test cases. This technique leverages the score values evaluated by MTK, heuristically motivated by the fact that same test scores are likely obtained by the same sets of

covered changed methods. Initially, the test suite is partitioned in *equivalence classes* of test cases with equal scores assessed by MTK. Then, the equivalence classes are sorted in descending order of scores, so that the sets of test cases which cover the highest number of structural changes are preferred. Finally, the prioritized test suite is organized by suitably picking a test case from each equivalence class following a round-robin policy. This process continues until all test cases have been re-arranged in a new permutation. The approach, namely MTK-QS, takes advantage of structural similarity to guess the relative coverage on changed methods of two test cases indirectly, without explicitly comparing the coverage set of the tests, which can be heavily time-consuming for large test suites. More formally:

Definition 4.9 *Given a test suite TS, and any pair of test cases $T_1, T_2 \in TS$, the relation $\equiv$ defined on TS is:*

$$T_1 \equiv T_2 \iff \text{score}(T_1) = \text{score}(T_2)$$

where score *is the function in Definition 4.8.*

The relation $\equiv$ is an *equivalence relation* on TS. In fact, the following properties hold:

- *reflexive*: for all $T \in TS$, $\text{score}(T) = \text{score}(T)$, thus $T \equiv T$.
- *symmetric*: for all $T_1, T_2 \in TS$, if $T_1 \equiv T_2$, then $\text{score}(T_1) = \text{score}(T_2)$, hence $T_2 \equiv T_1$.
- *transitive*: for all $T_1, T_2, T_3 \in TS$, if $T_1 \equiv T_2$ and $T_2 \equiv T_3$, then $\text{score}(T_1) = \text{score}(T_2) = \text{score}(T_3)$, so $T_1 \equiv T_3$.

As the defined relation $\equiv$ is an equivalence, it is therefore possible to partition the test suite TS in the *quotient-set* $TS_{/\equiv}$. Given a test $T \in TS_{/\equiv}$, the *equivalence class* $[T]_{\equiv} \in TS_{/\equiv}$ is the set of test cases with the same score values.

Exploiting the quotient set TS, the approach consists in re-ordering the test suite by picking test cases from these equivalence classes in a round-robin fashion, starting from those having a higher score. The intuition is that this kind of re-ordering can produce permutations with a faster and

larger coverage rate of methods which have been modified between the two software versions.

More in detail, the equivalence classes in $TS_{/\equiv}$ are initially sorted in descending order of scores. Then, a test case from the first and highest-scored equivalence class is picked and put in the first position in the prioritized test suite. The test is removed from its equivalence class when it is moved in the permutation. MTK-QS, in turn, selects a single test case from each subsequent equivalence class, appending it in the first empty position. After picking a test from the last equivalence class, the selection process is restarted from the first class. This round-robin selection process until all test cases have been picked and the final permutation has been completed.

Among the various different equivalence classes, the one containing test cases with a score equal to 0 should be treated differently. In fact, all test in this zero class do not cover any changed method, as the sums of dissimilarities according to score in Definition 4.8 is 0. As the rationale of MTK-QS is to give higher priority to test cases covering greater extents of changed code, the execution of tests in the zero class should be delayed. To do so, the MTK-QS approach exclude this class from the round-robin phase, and append its test cases in the last positions of the permutation after all other equivalence classes are emptied.

When selecting a test case from an equivalence class, several criteria can be chosen, for instance, by picking randomly a test in the class. MTK-QS employs a *total coverage* criterion to select the test case to pick inside a class. This criterion chooses the test case in the equivalence class covering the highest number of methods, both changed and unchanged. This criterion is more likely to increase also the rate of coverage of all methods in the re-ordered test suite.

An example of the execution of the MTK-QS is presented in Figure 4.9. All test cases of the suite have been already scored as in the MTK approach, and the background of test cases represents its score: the darker the background, the higher its score. A quotient set is there extracted according to relation $\equiv$, defining three equivalence classes. From the highest-valued class, the first test T_1 is removed from the class and placed in the first place of the permutation. Subsequently, the first test T_3 in the second equivalence class is selected, and moved to the second position. The next class is the zero class, which is ignored, and the following test will be picked

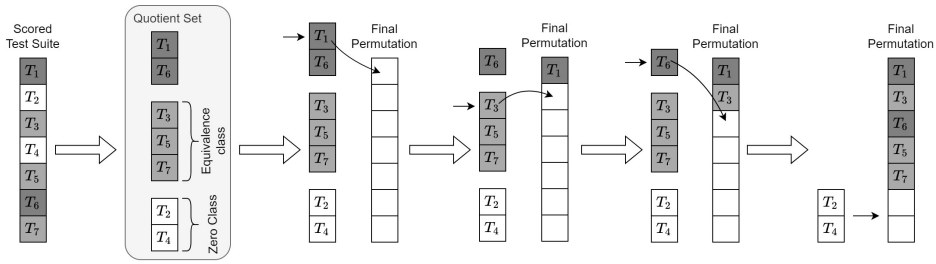


Figure 4.9. An example of the MTK-QS approach to rearrange test cases. The darker the background of a test case, the higher its score. White background means a score equal to 0.

again in the highest-value class again. In this case, T_6 will be inserted. After all test cases have been placed from all other equivalence classes, the test cases in the zero class are directly added to the last positions of the permutation.

The pseudo-code of MTK-QS is shown in Algorithm 4.4. It receives as input directly the quotient set of test cases. This can be readily obtained through the application of MTK, partitioning its produced permutations in the different equivalence classes. In the first place, the zero-valued equivalence class $[z]_{\equiv}$ is removed from the quotient set $TS_{/\equiv}$, as it will be appended after all other test cases have been inserted in the prioritized suite. Lines 2-3 implement the total coverage selection criterion, by sorting each equivalence class in descending order of method coverage. This makes test cases with higher method coverage being preferred among the others belonging to the same class. The loop in lines 6-13 is the core of this prioritization approach, and runs until all test cases in non-zero classes have been placed, i.e., the quotient set did not become empty. Lines 7-10 implement the loop which extracts a single test from each class, appending it to the prioritized suite and removing it from the equivalence class (line 8). The `if` block at lines 9-10 manages to remove empty equivalence classes from the quotient set, in order to eventually meet the `while` loop termination condition. After the loop is over, the zero class is appended as it is to the last positions of the permutation (line 14), which is eventually returned.

Algorithm 4.4 has been defined without explicitly calculating the score

Algorithm 4.4 Quotient-Set Prioritization.
Input: $TS_{/\equiv}$: quotient set containing equivalence classes of test cases with the same scores
Output: TS': prioritized test suite

```
1: sortEquivalenceClassesDescendingly(TS_/≡)
2: [z]_≡ ← getAndRemoveZeroClass(TS_/≡)
3: for each [T]_≡ ∈ TS_/≡ do
4:    sortTestCasesDescendingByTotalCoverage([T]_≡)
5: end for
6: TS' ← []
7: while TS_/≡ ≠ ∅ do
8:    for each [T]_≡ ∈ TS_/≡ do
9:       TS'.append([T]_≡.pop())
10:      if [T]_≡ = ∅ then
11:         TS_/≡.remove([T]_≡)
12:      end if
13:   end for
14: end while
15: TS'.appendAll([z]_≡)
16: return TS'
```

function, but it receives as input a test suite whose tests have been already scored. This separation underlines that any function which assigns a score to test cases can be inserted in place of score for relation $\equiv$. As the properties of $\equiv$ are not related to the particular shape of the score function, it is still an equivalence relation regardless of the specific function used to evaluate scores. It is thus possible to create different kinds of quotient sets according to different score criteria, as long that is possible to express these criteria through a function of test cases. This more general approach, namely QS, can be employed with every prioritization technique which ranks the test cases by assigning them a score. When the specifically applied to scores evaluated through MTK, the approach will be appointed as MTK-QS. In this case, QS has the effect of re-arranging test cases in order to provide a better rate of coverage of changed parts, based on the assumption that equal MTK scores indicate same sets of covered methods. However, in general different scores might produce permutations

with different properties and, possibly, vary the effectiveness of the underlying RTP technique.

4.4 Experimental Design and Empirical Setting

This section describes design and methodology used to set up the empirical evaluation of the MTK and MTK-QS effectiveness, and to compare it with different baseline techniques. The employed experimental setting is an extension of the study carried out for Genetic-Diff, presented in Section 3.3, involving projects written in Java and using code mutation to inject faults in subject projects, to obtain a broader set of software evolution-like scenarios. The entire dataset, along with the implementation of all techniques, is available in the replication package of the study [10].

Initially, this section describes the collection of subject projects and the process of fault injection. The second part of the section presents the RTP techniques which are used as baselines for the evaluation, and details of the implementation of all techniques. Eventually, the empirical setting used to evaluate the techniques is described, along with metrics used to evaluate performances and hypotheses of significance tests performed.

4.4.1 The Collected Projects

To evaluate the performance of the proposed techniques, a dataset of software projects, with different versions for each project, has been collected. The projects and the versions were needed to experiment the MTK and MTK-QS techniques in order to contextualize their execution in a software evolutionary scenario, according to the inputs to the pipeline described in Figure 4.8. As done in the experimentation of Genetic-Diff presented in Section 3.3, the project collection started from two well-known datasets of Java projects already employed in different RTP studies [82, 83]. However, the dataset collected for the Genetic-Diff experimentation has been extended, considering 5 different projects and a higher number of versions for each project. Furthermore, instead of relying on the original defined fault injection points as done in Genetic-Diff evaluation, a better procedure of fault injection has been carried out in order to cope with the coarser level of changes leveraged by MTK and MTK-QS. The extension

Project	Versions	LoC	Methods	Tests	Common	Changes	Description
AssertJ	6	62k	7985	5134	4994	62	Testing framework
JOpt	4	7k	860	639	532	14	Command-Line Interface parsing library
Joda-Time	4	160k	9412	4183	4045	28	Date and time processing library
La4J	6	9k	992	383	328	38	A linear algebra framework
Scribe	5	3k	344	74	61	12	A library for *O-Auth2* protocol

Table 4.1. Overview of subject projects used in the empirical evaluation of MTK and MTK-QS. *Common* indicates the average number of tests common to pairs of versions used in this study, while *changes* is the number of methods which have been changed from the previous version. Reported values represent the average across the considered versions.

in the empirical evaluation setting allows to analyze and discuss the performance of the proposed techniques on a wider and more heterogeneous set of benchmark scenarios, in order to provide a better generalization of results.

The complete list of projects, along with the number of versions collected for the study and some of their statistics are presented in Table 4.1. As it can be possible to see from the table, the projects differ in size, ranging from small projects with about 3k LoC and 300 methods in total, to large projects with more than 150k LoC and almost 10k methods. The number of test cases in the projects test suite also varies broadly, and is not always directly proportional to the size of the project. For example, the *AssertJ* project is smaller than *Joda-time*, but the number of its test cases is around 20% higher. The LoC values and number of test cases reported in the table relate to the average between all the versions involved in the dataset for each project.

Furthermore, as the main focus of RTP is to re-order the test cases to reduce the risk of software regression, Table 4.1 shows also, in the *common* column, the average number of test case that are present in both versions of the evolutionary scenario, aggregated by project. Usually, a permutation produced by any RTP technique is a re-ordering of these common test cases only, while different policies are applied to tests freshly added in the new releasing version and thus they are not prioritized. The table presents also the average number of changed methods between versions in each project, as this information is crucial for churn-based techniques.

Projects in the collected dataset have also different natures, and their application domain is sensibly different from each other. *AssertJ*[3] is a testing framework for the Java language with a rich set of assertions and a fluent syntax to rapidly produce more readable test cases. *Jopt*[4] is a Java library to parse command line arguments. These two projects have been employed also in the empirical evaluation of Genetic-Diffs, but their number of versions has been increased: from 4 to 6 for *AssertJ*, and from 4 to 5 for *JOpt*. *Joda-time*[5] is a Java library used to manage time and dates, and it was one of the most used libraries to deal with dates prior Java version 8. Its components have been actually integrated in the Java core libraries. *La4J*[6] is a Java framework to perform linear algebra tasks. It is currently not supported anymore, but it has been included in the study as it differs significantly from other projects in its developmental style, due to its nature of mathematical library. Finally, *Scribe*[7] is a simple *OAuth-2* client Java implementation. Note that the *Metrics* projects employed in the study of Genetic-Diff has been excluded, as the number of changes at the coarser granularity level of methods was too small and did not allow enough fault-injection points.

For each collected project, all versions have been labelled by their rank in the chronological order of their release. Thus, version 1 represents the oldest version collected for a project, while the n-th version, with n equal to the total number of version available for the project, is the last and most recent one.

Project in the original datasets presented in [82, 83] included no failure when executing the test suite. To this end, the methodology employed in these datasets leveraged on artificial faults, automatically injected in the software via code mutation. The original studies reported in depth the locations in which the faults were injected, and the same information has been used on the subset of projects used for the experimentation of Genetic-Diff in Section 3.3.3. However, these faults were located only on parts of the software which are common to all the versions considered in the studies, and even if two versions were close enough, and thus with

[3] https://assertj.github.io/doc/, visited on 16/01/2024.
[4] https://github.com/jopt-simple/jopt-simple, visited on 16/01/2024.
[5] https://www.joda.org/joda-time/, visited on 16/01/2024.
[6] https://github.com/vkostyukov/la4j, visited on 16/01/2024.
[7] https://github.com/scribejava/scribejava, visited on 16/01/2024.

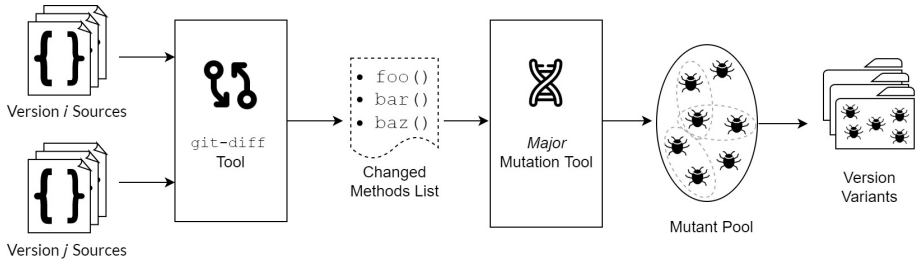


Figure 4.10. Visual schema of the mutant injection process for the MTK and MTK-QS empirical evaluation.

less changes between them, few injection points were available as several modules and classes are not present in all versions. When switching from techniques leveraging a statement granularity level to approaches employing a method granularity level, this reduced number of locations could limit the validity of the experimentation. Following this approach, in fact, even if two versions share a high number of methods which have been changed, several of these should not be designed to be mutated as these methods might not have been added yet in older collected versions.

For this reason, the fault-mutation process has been renewed, in order to supply a larger and more diversified pool of code mutations, which could be more representative of programming errors, for a specific pair a versions considered as an evolutionary scenario for the software. More specifically, given two software versions, i.e., a previous version V_i and current versions V_j, the points in which faults are injected are located in methods belonging to the code churn between V_i and V_j. After localizing methods in which faults should be injected, the fault seeding process and the creation of faulted variants proceed as in [82] and in the empirical setting of Genetic-Diff.

Figure 4.10 summarizes the process of fault-injection for the empirical evaluation at the foundation of experiments. For each pair (V_i, V_j) of chronologically subsequent versions in the collected dataset, methods which have been changed in at least one of their statements[8] are produced through the application of the `git-diff` tool[9]. This tool is part of the *Git*

[8]Thus, excluding methods whose changes are located in comments only.
[9]`https://git-scm.com/docs/git-diff`, visited on 16/01/2024

utility[10] and it is implicitly used by this Version Control System (VCS) to evaluate modifications between files and functions between two different *commits*. Among the options of `git-diff`, it is possible to refine the reports of changes between the versions to a function/method level. This granularity level has been used to create the list of changed methods between versions V_i and V_j.

Once the list of changed methods has been produced, similarly to the Genetic-Diff setting, the *Major mutation* tool[11] has been applied to produce code mutants for methods in the current version V_j on the list. All mutants have been placed into a mutant pool, and then 100 sets of five mutants have been extracted uniformly at random. For all these sets, the source code related to the mutant has been injected in the respective method of the original current version V_j, producing 100 variants of this version. The set of the 100 variants produced for a version will be referred as the *faulted version*. Each variant represents a scenario of software evolution in which different faults have been introduced. After variants have been created, a *fault matrix* for each variants has been produced. The matrix links all test cases with the faults they uncovers in a specific variant.

Note that the variants have been produced for each pair (V_i, V_j) of subsequent versions, with V_i and V_j not necessarily immediately subsequent, i.e., for all pairs in which $i < j$. This means that the first and oldest version V_1 of each project has no variants, while the last one V_n has a number of faulted versions equal to $n - 1$, one for each prior version.

4.4.2 Baseline Techniques and Implementation

To empirically analyze the performance of MTK and MTK-QS, a set of baseline RTP techniques has been chosen. These techniques are commonly employed in practice in many test prioritization scenarios, both in industry and in academic researches. Three baselines have also been employed in the Genetic-Diff study presented in Section 3.3.2, and will be briefly resumed in this section, while other two baselines have been introduced specifically for this empirical evaluation. The baseline techniques employed in this study are:

[10] https://git-scm.com/, visited on 16/01/2024.
[11] https://mutation-testing.org/, visited on 16/01/2024.

- *Total*: a greedy technique which scores each test cases with the total number of code units it covers [114]. To align this technique to MTK and MTK-QS, the granularity of code elements considered by the technique has been set to the method level. This technique has been included due to its popularity and to its good results in many practical cases.

- *ART*: an adaptive random prioritization technique, which constructs a permutation by selecting test cases maximizing the minimum distance, in terms of coverage, from those already inserted [67]. As with the total baseline, the coverage is considered at method-level granularity. It has been included among the baselines to have a non-deterministic RTP approach to compare the proposed techniques.

- *Genetic-APTC*: a GA with the APTC fitness function, with PMX crossover and Swap Mutator. The granularity of the fitness function is at method-level. As ART, this approach has been included to have a comparison with one more non-deterministic prioritization strategy.

- *Difference-Based Prioritization (diff)*: a RTP approach which scores test cases according on the number of changed code elements they cover. After scoring the test cases, the test suite is re-arranged in descending order of these scores. This technique prioritizes test cases according to the total number of changed methods a test covers. It employs a simple and naïve measure of differences, through the `git-diff` utility. Specifically, the changes of each method are evaluated in a Boolean fashion, whether the method has been modified or not during the evolution. This technique has no sensitivity on types of changes and employs a coarser evaluation of the code churn. This technique has been introduced among baselines to compare the impact of different change-measures in the experimentation.

- *Difference-Based with Quotient-Set Prioritization (diff-QS)*: a QS approach using the diff baseline as score function. In this case, the produced equivalence classes contain all test cases which cover the same number of changed methods. This technique has been introduced to analyze the performance of QS approach using a different

score function with respect to MTK-QS, and to state whether the combination of MTK and QS might produce more benefits for RTP than other code-churn measures.

Aside from the baselines listed above, MTK and MTK-QS have been also compared with the test suite in its original order, defined by developers or by the *test runner* utility. The permutation with test cases in this original order is defined as the *untreated* permutation, and it is referred to as the No-Prioritization (NOP) technique.

To realize the MTK and MTK-QS techniques, an implementation of the inner PTK function used to evaluate similarity between ASTs had to be performed. To this end, the efficient and robust implementation of Tree Kernels available in the *KeLP* framework[12] has been used. *KeLP* (*Kernel-based Learning Platform*) [46] is a Java framework focused on Machine Learning approaches based on kernel functions. It supports different types of convolution kernels, TKs among them, and offers a wide variety of formats to serialize and deserialize tree structures. *KeLP* has been extensively used in several different studies in the field of NLP, but it can be readily adapted to work on tree representations involved in other branches of computer science, such as Software Engineering.

The MTK-QS implementation has been split in two modules: the first directly uses the implementation of MTK to score test cases, while the second module accepts the collection of scored test cases to construct the quotient-set and therefore applies the operations defined in Algorithm 4.4. This separation of concerns allowed to implement also diff-QS using the same code of the diff strategy, along with this more general implementation of QS. Concerning the other baselines, the implementation provided within the studies on the original dataset [82] has been used, with some tiny modifications to their code in order to embed the techniques in the prioritization pipeline set up for the experiment. The source code for all techniques involved in the empirical evaluation can be found in the replication package [10] of the paper in which MTK and MTK-QS have been proposed.

[12] http://www.kelp-ml.org/?page_id=728, visited on 18/01/2024.

4.4.3 Empirical Setting for Evaluation

To apply the proposed RTP techniques, the setting for the empirical evaluation has been designed. RTP strategies are part of regression testing activities, which are performed in scenarios of software maintenance. To this end, each experiment should resemble one of these scenarios, and involves the evolution from a *previous* and older version to a *current* version in releasing, which is under test and whose test cases need to be prioritized. The information required by all techniques is assumed to be available on the previous version, and it is used to drive the prioritization process according to the specific measures considered by different approach.

The considered churn-based approaches, i.e., MTK, MTK-QS, diff and diff-QS, need to quantify the extent of changes in the code. This evaluation can be performed starting by the original source code of the previous and the current versions, which are typically available in many practical cases, and therefore modifications can be quantified according to the specific approach of techniques. These approaches make use of per-test coverage reports to join code churn information with test cases exercising changed parts, while the other baselines use code coverage directly to build the permutations. In any experiment it is assumed that per-test code coverage reports are present for the previous version, and used to approximate the coverage of test cases in the current version.

Each experiment in the empirical evaluation can be described by a pair of versions, the first one representing the starting and *previous* version in the evolutionary scenario, and the specific variant of the *current* version under test. An *experimental pair* denotes each scenario, and can be denoted with $(V_i, V_{j,k})$. V_i represents the previous version, which is evolving into version V_j, with the original source code aside for mutants included in the group of faults for the k-th variant. The order of version is relative to their chronological ordering in the set of collected versions for a project. The experiments cover all possible evolutionary scenarios with each project, plausible pair of versions and its 100 variants available. In particular, the set of all experimental pairs for a project can be defined as $\{(V_i, V_{j,k}) | 1 < i < j \leq n \wedge 1 \leq k \leq 100\}$, where n is the number of versions collected for the project (and reported in Table 4.1). For a pair to be plausible, the versions involved should enforce the constraint $i < j$, i.e., the starting version should be older than the other. Although it is possible

to also consider pairs not verifying this constraint, the resulting scenario might not be representative of a real case of software evolution, and therefore pairs of this type have not been considered in the experimentation.

To compute the total number of different pairs of versions involved in the experiment, if n is the number of versions for a specific projects, all the experimental pairs are $\frac{n \cdot (n-1)}{2} \cdot 100$. In fact, version V_1 is never considered as the current versions, V_2 is involved in only pairs with V_1 as previous, V_3 only in pairs with V_1 or V_2 as previous version, and so on. As for each pair of versions there are 100 variants of the current V_j, the total is multiplied by 100. This leads to 52 different arrangements of versions in pairs, and thus a total of 5200 experimental pairs, on which all techniques were executed.

The main goal of the empirical evaluation of MTK and MTK-QS is to assess their fault-detection performance on the collected dataset, in order to compare results with the widely-used state-of-art baselines. A first analysis on the permutations produced by the proposed techniques and the baselines employed the evaluation of the APFD metric, presented in Definition 2.2, to assess the rate of fault-detection on experimental pairs composed from the dataset. To assess whether statistically significant differences exist in results obtained by the APFD evaluated between the two proposed techniques and the baseline approaches, a *Wilcoxon-Mann-Whitney U-test* has been evaluated. The test presents the following *null hypothesis*:

Definition 4.10 *For each subject project s, and RTP techniques t_1, t_2, the* null hypothesis *is:*

$\mathbf{H_0^{t_1,t_2,P}}$*: The APFD value of technique t_1 on project p is not greater than the APFD value of technique t_2 on the same project.*

Moreover, to quantify the extent of resources which might be saved by the application of the different techniques, the PFF and PLF metrics, respectively shown in Definition 2.4 and 2.5, have been calculated for all experimental pairs. These metrics give insights on which fraction of the test suite permutation should be executed, for each technique, before one or all faults have been discovered.

The fault-detection analysis is carried out on two levels of aggregation: on a per-project basis, in order to discuss the fault-detection performance

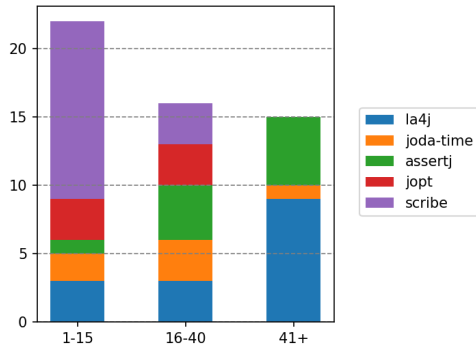


Figure 4.11. Distribution of experimental pairs according to the number of changed methods between the previous and current versions. Versions belonging to same projects have same colors.

of techniques according to the nature of the project on which they are executed, and on the magnitude of changes between two versions of experimental pairs. This latter analysis can help to inspect the fault-detection behaviour of the proposed techniques on projects with different characteristics when changes of different magnitude are applied. To this purpose, the pairs of versions in the dataset have been partitioned in three intervals, guided by a frequently employed characterization in the software effort estimation domain [127]: a *small* change interval, which includes pairs of versions for which the number of changed methods range from 1 to 15; a *medium* interval, containing pairs with 16 to 40 changed methods; and a *large* interval, with versions with more than 40 changed methods.

The distribution of pairs of versions in the dataset, according to their respective bin, is presented in Figure 4.11, highlighting the project they belong. As can be seen, the pairs are nearly equally distributed between intervals. All bins include pairs from almost all projects, with the exception of the smallest projects *JOpt* and *Scribe*, which due to their limited size have no pair in the large bin. Even larger projects such as *Joda-Time* and *AssertJ* present pairs with heterogeneous degrees of changes, from new releases with small modifications to large patches bringing a high amount of changes between previous and the current versions.

MTK and MTK-QS perform an evaluation of the extent of changes to the source code when the software moves from a previous version to a new version. The steps, resumed in Figure 4.8, involve several operations, from the construction of the ASTs related to methods, their matching between the versions, the evaluation of the PTK dissimilarity, the sorting of the test suite and, in case of MTK-QS, the re-arrangement of the permutation. The pipeline of these operations is more time-consuming compared to the considered baseline techniques. In order to account to what extent the operations performed by MTK and MTK-QS are more costly, in terms of time, than the other techniques, an analysis of the execution time has been performed.

To perform benchmarks on the execution time of all techniques, the Java Micro-benchmark Harness (JMH)[13] framework has been employed to collected times of the execution of all techniques. JMH provides tools to ease the definition of benchmarks in the Java programming language, allowing to run them in different configurations, and employing different independent instances of the Java Virtual Machine (JVM) to increase the reliability of results.

The execution time analysis has been performed on the pair of versions with the greatest number of changed methods, in the worst scenario both for the proposed Tree Kernel-based techniques. To reduce noise and inconsistencies in the collected data, each technique has been repeated 30 times, with two independent instances of the JVM. The time to load the coverage reports have been excluded from the evaluation, as all techniques rely on this information. The time related to the detection of changed files between the pair of versions in an experiment has been excluded as well, as in practical scenarios this information is directly evaluated by the VCS used for the subject projects when the new version is submitted to the codebase, and thus it is already available at no cost.

Usually, the most common constrained resources in RTP scenario is time. An analysis of the time saved by MTK and MTK-QS has been thus performed using the EET_r metric presented in Definition 2.6. This measure can help to quantify the amount of time required by a prioritization technique both to construct the permutation and to execute the fraction of test cases needed to uncover all faults, relatively to the total time con-

[13] https://github.com/openjdk/jmh, visited on 16/01/2024.

sumed by executing the entire suite. The results for MTK and MTK-QS have then been compared to those obtained for the baselines, in order to have insights on the benefits on their usage in regression testing activities.

To assess also the statistical significance of EET_r results, two *Wilcoxon-Mann-Whitney U-test* have been performed, to check if obtained results for different techniques on different experimental pairs originate from lower or greater distributions. In this case, a lower distribution is better, as it identifies a technique with lower EET_r and, thus, saves more time in the validation phase. The two tests have the following null hypotheses:

Definition 4.11 *Given a project p and two RTP techniques t_1 and t_2, the null hypotheses are:*

$\mathbf{L_0^{t_1,t_2,p}}$*: The effective execution time of technique t_1 on project p is not shorter than the effective execution time of technique t_2 on the same project.*

$\mathbf{G_0^{t_1,t_2,p}}$*: The effective execution time of technique t_1 on project p is not longer than the effective execution time of technique t_2 on the same project.*

The confidence limit, α, has been set at 0.05. To mitigate the *multiple comparison problem*, arising when multiple hypotheses are tested, a standard Bonferroni correction [25] has been used, as commonly employed in other software engineering studies [118, 92].

4.5 Results and Discussion

This section illustrates the results of the experimental evaluation of MTK and MTK-QS techniques and their comparison to the baseline approaches. The effectiveness of the proposed prioritization techniques on collected projects has been analyzed using fault-detection metrics and in terms of their execution time. The analysis of fault-detection performances has been performed both according to the characteristics and nature of subject projects, and in relation to the number of changes between the pairs of versions considered in experiments. The execution time of different techniques have been performed on a project basis, firstly measuring the time spent by the considered approaches to produce a permutation, and the using then EET_r metric to quantify their practical effectiveness.

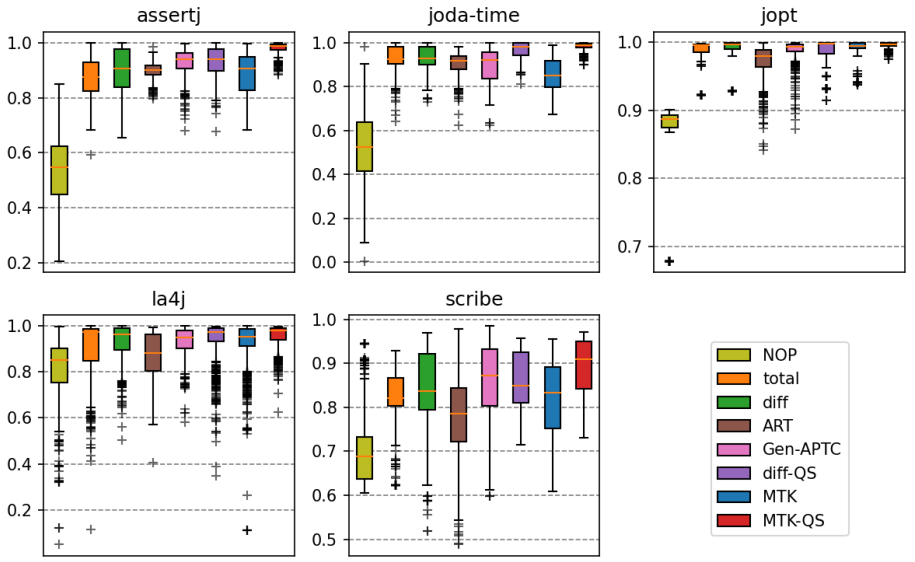


Figure 4.12. Box-plots of resulting APFD values for MTK, MTK-QS and the baseline techniques, aggregated on projects.

The first section presents the results of the empirical evaluation related to the fault-detection performance, along with a discussion on the obtained results. The second section details the analysis of the time to execute the techniques and the extent of resources spared by the application of MTK and MTK-QS with respect to the baselines. Finally, some threats to validity, limiting the generalizability of the evaluation, are presented.

4.5.1 Fault-Detection Performance

For all evolutionary scenarios included in the empirical setting, MTK and MTK-QS have been executed, along with the other considered baselines. For each pair of versions, the techniques were applied to all variants in the pair. along with the baseline techniques. All variants with injected faults of the last version have been considered, and APFD, PFF and PLF have been evaluated for each permutation produced by these techniques.

APFD results for the analyzed techniques are shown in Figure 4.12. The results of the various techniques have been aggregated on a project

basis, and presented as boxes in the figure. More specifically, each technique has been executed on all pairs of version of a specific project, and on all the variants in the pair. The APFD value has then been evaluated for every permutation produced, and all these values were aggregated together and depicted by the respective box in the plot. As it can be seen, any prioritization approach significantly improves the rate of fault detection with respect to the untreated test suite (labelled with NOP). Although in two of the subject projects (*JOpt* and *La4J*) the fault detection rate of the untreated test suite is high (typically around 0.85), all prioritization techniques exhibit a significant increase in the average rate of fault detection.

The APFD results for MTK are generally lower than the baselines, or at most comparable to them, for quite all projects. The low scores obtained were due to the redundancy in the coverage of changed methods in the scheduling of test cases. In fact, several tests which cover the same changed methods, are assessed with the same score by MTK, and are therefore placed one after another in the output permutation. As a consequence, other changed methods in the version are exercised only after the execution of a (possibly) long sequence of test cases with the same score. This has the effect of reducing the fault detection performance of the permutation, leading to poor results.

As a practical example, Table 4.2 shows the resulting permutation of MTK for the experimental pair (V_1, V_2) of the *Joda-time* project, on one of its variants. The table includes the rank of a slice of test cases, with test names, the scores assigned by MTK and the faults discovered by each test. As highlighted in the table, the first three tests have the highest score, and are hence scheduled before the other test cases. According to coverage reports, these three test cases cover the same changed methods[14], and belong to the same test class. None of these tests, however, discover any fault, as in the considered variant they only cover methods which were not injected by any mutation. As a consequence, the earlier scheduling of these test cases does not increase the permutation fault-detection rate. The fault F_1 in the variant is discovered by the 4-th test case, which has a lower score than the previous ones as it covers a lesser extent of changes. The subsequent test cases all cover exactly the same changes and discover the same fault F_1, so their execution, one after another, penalizes the APFD

[14] As their name also suggests...

Rank	Name	Score	Faults
1	`testProvider`	0.735	$\emptyset$
2	`testProvider_badClassName`	0.735	$\emptyset$
3	`testNameProvider_badClassName`	0.735	$\emptyset$
4	`testConstructor_Object5`	**0.622**	$\mathbf{F_1}$
5	`testConstructor_RI_RD1`	0.622	F_1
⋮	⋮	⋮	⋮
19	`testAdd_RInterval4`	0.622	F_1
20	`testToInterval`	**0.526**	$\mathbf{F_2}$
21	`testToInterval_nullZone`	0.526	F_2
⋮	⋮	⋮	⋮

Table 4.2. An example of a permutation produced by MTK for an experiment on *Joda-Time*. Each row is related to a test case. The *Score* column shows the score assigned to a test case by MTK, while the *Faults* column reports the set of faults the test case in the row discovers. Rows in bold highlight the first test case which discover a new fault. Horizontal rules in the upper table shows sets of test cases with the same score.

value of the permutation again. The next valuable test case is in position 20, with a different score from previous tests, and it uncovers the new fault F_2, positively contributing to the APFD value of the permutation. Other test cases are assessed with the same score, and scheduled to be executed after it until new faults are discovered by different test cases with different scores. This phenomenon repeats until the end of the permutation.

This behaviour has a particular impact on projects with a high number of test cases, such as *Joda-time* and *AssertJ*, or when the number of changes is limited. In these scenarios, it is more likely that many test cases exercise the same changed methods. This occurs even more often if the changes are highly distributed in core modules of the software. Methods in these modules are typically called from several other components, which might have remained unchanged in the evolution. However, test cases in these components are likely to cover the changed methods in core modules, thus being entitled with the same score and scheduled one after another even if previous tests already exercised the core changed methods. Due to this, the risk of depleting a significant amount of resources allocated for the test

Rank	Name	Score	Faults
1	`testProvider`	0.735	$\emptyset$
2	`testConstructor_Object5`	**0.622**	$\mathbf{F_1}$
3	`testToInterval_nullZone`	**0.526**	$\mathbf{F_2}$
4	`testPatchedNameKeysLondon`	0.482	F_2
5	`testWithers`	0.478	F_2
⋮	⋮	⋮	⋮
110	`testProvider_badClassName`	0.735	$\emptyset$
111	`testConstructor_RI_RD1`	0.622	F_1
112	`testToInterval_nullZone`	0.526	F_2
⋮	⋮	⋮	⋮

Table 4.3. An example of a permutation produced by MTK-QS for an experiment on *Joda-Time*. Each row is related to a test case. The *Score* column shows the score assigned to a test case by MTK-QS, while the *Faults* column reports the set of faults the test case in the row discovers. Rows in bold highlight the first test case which discover a new fault.

phase due to repeatedly stressing the same code units is increased.

On the other hand, the *Scribe* project has limited size and a small number of changes between versions. In this case, the majority of test cases are assessed with a score equal to zero, while the others are divided into a few small groups with the same score. This phenomenon is enhanced also due to the small number of test cases in the test suite of the project, and delays in discovering faults penalizes the permutation APFD value significantly more than in other projects.

In the experimentation, MTK exhibited fault-detection performances worse than all the baselines, due to its redundancy issue. Unfortunately, the experimentation showed that the MTK technique does not provide any benefit compared with other prioritization techniques. It might not be successfully applied to the RTP problem without some device to correct its redundancy issue.

According to the experimental APFD results depicted in Figure 4.12, the QS applied to MTK limits and drastically enhances the fault-detection performance of the technique, as it can be seen from the fault-detection

performance of MTK-QS. In fact, picking in turn from equivalence classes reduces the chance that subsequently-scheduled test cases exercise precisely the same changed methods. To this end, Table 4.3 presents the prioritized test suite generated by MTK-QS in the same setting as the MTK example in Table 4.2. The first scheduled test is the same for both techniques, as it belongs to the equivalence class with the highest score and it also possesses the highest total method coverage in the class. Unfortunately, it does not discover any fault. However, in this case, the next test chosen by MTK-QS belongs to the second, higher-scored equivalence class, which MTK placed in the fourth position. MTK-QS thus discovers the first fault F_1 with the test case scheduled in the second position. Furthermore, the third test case in the permutation uncovers a new fault, F_2, quicker than happened with MTK. After a test case has been picked for each equivalence class, the process restarts again from the first and high-ranked one, as can be seen with the test in position 110.

As shown in Figure 4.12, MTK-QS outperforms both MTK and the other baselines on all projects, with diff-QS being the baseline whose APFD results are closer. Although QS applied to changes evaluated by textual differences often leads to better performances in comparison with the diff prioritization alone, the MTK-QS approach typically produces test suite permutations with a higher rate of fault detection. This is due to the difference in the score functions between the two methods: a score based on the number of the changed methods covered by a test case does not suffice to discriminate between methods covered by the test, and equivalence classes defined on it have a smaller chance to exercise the same changes than in MTK-QS. This lead to the risk of scheduling test cases covering more changes later in the permutation.

On the other hand, QS applied on more refined churn-based scores can lead to better diversification on coverage of changed methods. This suggests that the score assigned by MTK can be seen as a kind of *fingerprint* of the set of changed units covered by a test case. The diversification of coverage exhibited by MTK-QS is higher even compared with Genetic-APTC and ART techniques, which search for a permutation with an increased coverage rate in the set of all permutations. In all scenarios, indeed, the lack of exploitation about changes in these techniques leads to worse APFD performances.

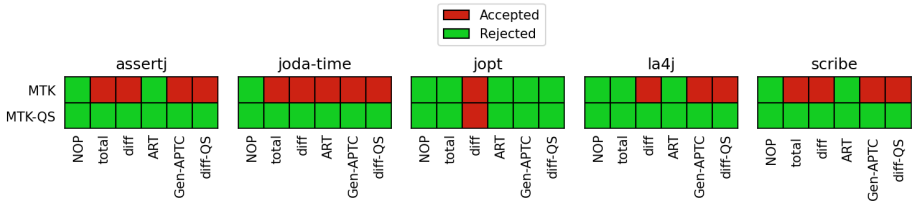


Figure 4.13. Results of Wilcoxon-Mann-Whitney U-Test on APFD results, for the statistical significance of the proposed techniques, with respect to the baselines, aggregated on projects. The color of the cells means that the null hypothesis can be accepted (red) or rejected (green) between the technique on the row and the one on the column.

Figure 4.13 shows the results of the Wilcoxon-Mann-Whitney U-test of the proposed techniques with respect to the baselines, after the application of Bonferroni correction, with the *null hypothesis* $\mathbf{H_0^{t_1,t_2,p}}$ presented in Definition 4.10. Compared to all the baselines, it is not typically possible to reject the *null hypothesis* for the MTK technique, and in the majority of cases there is no significant improvement in the fault-detection rate of the produced permutations by this approach. MTK-QS, on the other hand, constructed prioritized test suites with higher fault-detection rates than the baselines in almost all subject projects, and the null hypothesis can be rejected with a high amount of confidence (typically, $p \ll 0.05$ in almost all cases). The only exception to this general trend is the *JOpt* project, where MTK-QS produces prioritization whose fault-detection rates are not significantly greater than the *diff* prioritization. In this particular project, however, all techniques involved in the study produced APFD values highly close to 1, which makes almost all the produced permutations similar to each other in terms of fault detection.

In general, the experimental results suggest that MTK-QS tends to perform statistically better in terms of fault detection rate than the other techniques considered in the study.

Figure 4.14 shows the aggregation of APFD values obtained on experimental pairs in three bins, according to the number of changed methods between versions in the pairs. The first bin is related to small changes (1-15 changed methods), the second to medium changes (16-40 changed

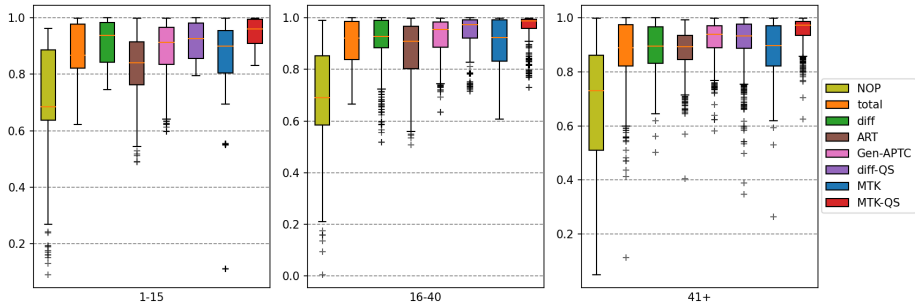


Figure 4.14. Results of analysis related to the number of changed methods between versions in an experimental pair. On top, APFD values for subject projects aggregated in small (1-15), medium (16-40) and large (41+) numbers of changed methods. On bottom, the distribution of experimental pairs in intervals of changed methods, divided by projects.

methods) and the third to large changes (more than 40 changed methods). MTK results show that, in any bin, the approach has worse performances with respect to the other baselines. Even changing the aggregation of data, the redundancy issue reducing the coverage rate of permutations is still overwhelming. This happens on all change entities: in fact, when the changed methods are few, the risk that MTK assigns the same score to test cases is high, and as previously discussed, the exercising of changed methods not yet uncovered is postponed. Conversely, as the number of changes increases, the same does the variety of scores of test cases. However, test cases which cover different sections of changed methods have also a smaller probability of exposing the faults. In this case, several tests covering non-faulted methods could be executed before subsequent cases eventually show the other faults.

Results obtained by executing MTK-QS can also justify these observations on MTK. In fact, in all experimental scenarios, MTK-QS outperforms MTK, obtaining the highest APFD values precisely on pairs with the largest number of changes. Fault-detection performances of MTK-QS are also better than all baselines, both aware and unaware of changes, also for all extents of changes. From this analysis it is also highlighted that the improvement in APFD values of MTK-QS with respect to MTK is higher than the application of QS to the diff technique. In particular,

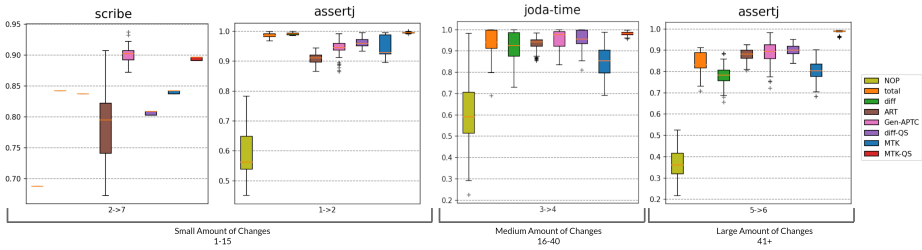


Figure 4.15. APFD results for specific experimental pairs, subdivided in small (1-15), medium (16-40) and large (41+) intervals of number of changed methods.

when considering small projects, the diff-QS performs slightly worse than diff. In scenarios in which there is a small number of changed methods, it is highly likely that several test cases have low diff scores, narrowing the score distribution among tests. In these cases, the round-robin policy of QS is responsible of a delay in the scheduling of test cases which may actually cover a larger number of changed methods. As the number of changes increases, this phenomenon is reduced due to a wider distribution of diff scores of test cases. This suggests that the scores produced by MTK are more meaningful to be combined with the re-arrangement induced by QS. In fact, MTK-QS produces permutations with higher rates of fault detection, simultaneously exhibiting a lesser degree of variance and thus proving more stable than the other approaches.

To further drill down the analysis of the proposed techniques, Figure 4.15 presents a finer analysis, focused on four specific experimental pairs. The presented APFD values have been aggregated on all the 100 variants of each pair. The considered pairs are labelled by the name of the project they belong and the relative order of versions occurring in the pairs. The pairs included in the figure are (V_2, V_7) of *Scribe*, (V_1, V_2) and (V_5, V_6) of *AssertJ* and (V_3, V_4) of *Joda-time*. Two pairs have been chosen from pairs in the interval which includes a small number of changed methods, while one pair has been extracted for both the medium and the large change-interval. As it can be noted from the figure, version pairs extracted from the small bin produces narrower boxes, due to the fact that a small number of changed methods introduced few point in which faults were injected, and groups of faults in variants tend to be more homogeneous. Only Genetic-APTC and

ART produce wider boxes, due to their non-deterministic nature.

As it is possible to see from the plots in Figure 4.15, diff-QS performs worse than the simple diff approach in both pairs with a small number of changes, while MTK-QS has significantly higher APFD results than MTK. When analyzing in detail the results on pair (V_2, V_7) of *Scribe*, it is possible to see that the ART technique outperforms all others on average. In this case, the variants in this particular pair are grouped in a manner which favorites the technique, which in general has bad fault-detection performances on the whole project. This also suggests that the analysis of single experimental pair out of a wider context can easily lead to mistaken conclusions, particularly for projects with a small amount of changes in which faults are homogeneous between variants.

Considering the experimental pair with medium and large changes, i.e. *Joda-Time* and the *AssertJ* pairs, MTK typically performs worse than the other techniques again. On the other hand, MTK-QS outperforms all techniques, and has higher APFD values even on the variants in which it performs worse. Furthermore, *AssertJ* and *Joda-time* are the largest projects in the collected datasets, both in size and in number of test cases (see Table 4.1). According to its results, in line with the aggregate case presented in Figure 4.12, the scheduling of test cases produced by MTK-QS can remarkably benefit the regression testing activities, allowing to rapidly discover a large amount of faults in these large-sized software.

To analyze how many resources can be saved by employing the proposed techniques compared to the baselines, the PFF and the PLF metrics have been evaluated on all scenarios in the experimental setting. Results for these metrics are depicted in Figure 4.16. For each technique and each project, two bars have been drawn: one, in green, with the average PFF of all versions and variants in the project, and the other, in blue, with the average PLF measure obtained in the same manner. The PLF bars include also the standard deviation error atop, as the resulting values were more spread around the average than PFF.

As it is possible to see from Figure 4.16, all techniques have typically low PFF values, with the exception of the untreated test suite (NOP). This means that permutations produced by any technique discover the first fault after the execution of a very small number of test cases (usually, less than the 5% of test cases). Thus, the application of any RTP technique

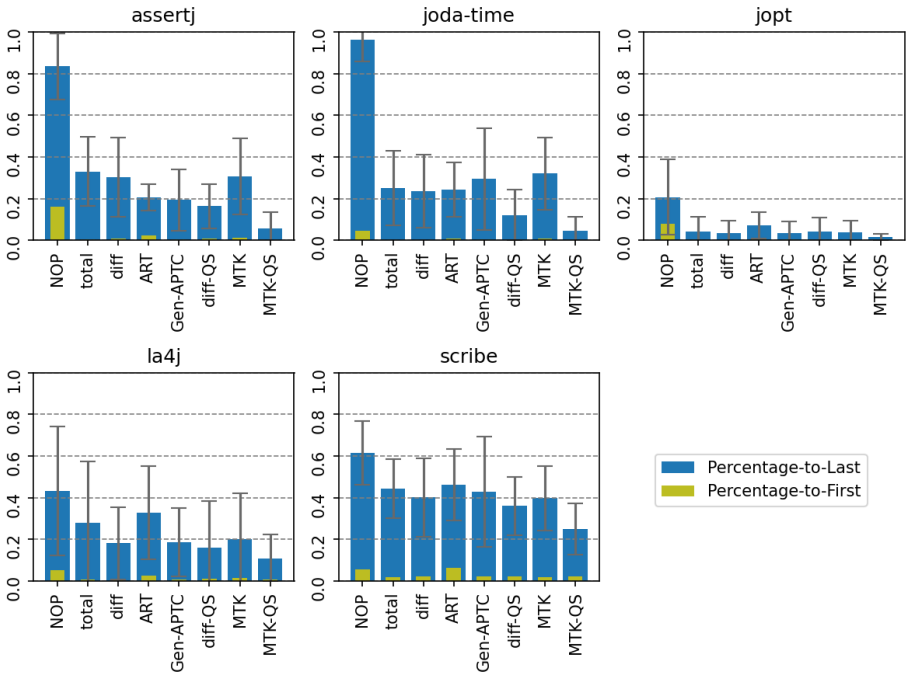


Figure 4.16. PFF and PLF values for the empirical evaluation of MTK and MTK-QS, aggregated on projects.

can produce benefits to the testing activities for the subjects, according to what has been observed for the APFD metric.

Conversely, PLF results are more various between the different techniques. Analyzing the results of MTK in depth, the PLF metric is higher than the other techniques in almost all cases, meaning a higher number of test cases to be executed before uncovering all faults introduced in the new version. This is due to the poor distribution of coverage of changed methods, and the application of MTK-QS gives significantly lower values. Instead, MTK-QS performs better than all the other techniques, discovering all the faults in the software with a small percentage of executed tests. The maximum PLF value for MTK-QS is obtained in the *Scribe* project, and it is around to 22%. This means that, on average, less than a quarter of test cases in permutations produced by MTK-QS needs to be executed

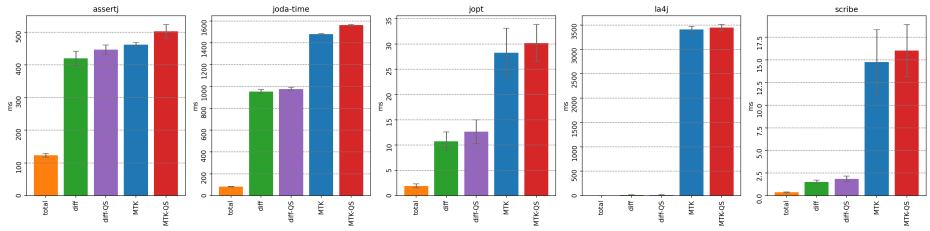


Figure 4.17. Average results for the execution time of MTK and MTK-QS, compared with baseline techniques, per subject project. The average is on 30 repetitions for the pair of versions of each project with the highest number of changes.

in order to fully uncover all faults in the version. The *Scribe* project is however small and has few test cases, thus every executed test case before discovering all faults accounts for a higher increase in the PLF metric. When considering other projects, it can be seen that the fraction of the test suite which needs to be executed to discover all the faults is on average below the 10%. Focusing on PLF results on *AssertJ* and *Joda-time* specifically, the percentage of the test suite which needs to be executed is on average less than 5%. As these projects are the largest ones among the dataset, this has the consequence of saving a higher amount of resources in their regression testing activity.

4.5.2 Analysis of Execution Time

Benchmarks to analyze the cost to execute the proposed MTK and MTK-QS techniques, along with all the other baseline, have been executed on the subject projects. The results of the benchmark for ART and Genetic-APTC baselines registered execution times that were typically 2 orders of magnitude higher than those of all other techniques, including MTK and MTK-QS. Due to their long times to produce a permutation, ART and Genetic-APTC were not applicable in practice to the subject projects in the datasets. To obtain reasonable execution times, different hyper-parameters (such as number of generation and population size for Genetic-APTC) have been highly reduced. However, permutations produced by the algorithms in these condition exhibited very low fault-detection performance on average, and were generally comparable to the

untreated test suite. For this reason, ART and Genetic-APTC baseline techniques have been excluded from this analysis.

Figure 4.17 shows the results of benchmarks on the subject projects in terms of execution time. As expected, the amount of time for the execution of all strategies varies according to both the project size and the number of test cases. Permutations for smaller projects, such as *Scribe* and *JOpt*, have been constructed in the range of few to few tens milliseconds, while for larger projects, such as *Joda-Time* and *AssertJ*, the execution time is significantly higher.

For three out of the five projects, i.e. *AssertJ*, *JOpt*, and *Joda-Time*, the execution time of MTK and MTK-QS is slightly higher than the baselines, introducing only little relative overheads. It can be observed that the evaluation time of structural similarity of changed methods using TK functions depends only on the number of changes between two software versions, and not directly on the size of the project. Considering the *Scribe* subject project, the execution time of MTK and MTK-QS is one order of magnitude higher. This difference is due to the small size of the test suite of the project, which causes the time for the evaluation of similarity to dominate both scoring and sorting steps common also to other baselines. However, even in this case, the execution time for the proposed techniques is still lower than 20 milliseconds, thus being affordable for prioritization purposes with respect to the whole execution time of its test suite.

A different scenario arises from the results for the *La4J* project. The execution time of the proposed techniques is, in this case, from two to three orders of magnitude higher than the baselines. The main cause of this is not ascribable to the size of the project, as *La4J* has about 9,000 lines of code, nor to the number of test cases in its suite, less than 400. The reason of these poor performances has to be found in the nature of the project itself. *La4J* is a library of linear algebra for Java and, for performance reasons, when writing its code, developers preferred *parataxis* to *hypotaxis*, i.e., long sequences of statements on the same level rather than nested blocks or auxiliary function calls. In particular, two methods of *La4J*, namely `decompose` and `hqr2`, are 465 and 462 lines of code long, respectively. Methods of this kind give rise to ASTs with large branching factor, and the impact on computation time for evaluating TK functions is exceptionally high (around 1 second for each method). For this reason,

MTK and MTK-QS might not be suitable for the *La4J* project, or other projects which share these characteristics. However, according to good practices of modern developmental methods, *La4J* is atypical, and similar scenarios should seldom be found in practice. It is indeed a common practice, during software development, to not exceed 20 lines of code in a method [87], to enhance the maintainability of the software. *La4J* does not follow these practice due to its particular nature of mathematical library. Other projects present long methods, e.g., *AssertJ* and *Joda-time* contains methods up to 200 lines of code long. These methods are however processed in few milliseconds, due to the presence of nested blocks limiting the branching factor of their ASTs and thus can be quickly evaluated by TKs.

It can also be seen that the application of QS prioritization, either using MTK and diff scores, introduces a negligible overhead. The partitioning the test suite in equivalence classes, and the round-robin selection of test cases within is straightforward if the test suite has been previously sorted by descending order of assigned score. In this case, in fact, both construction of the quotient set and the selection can be performed in linear time on the number of test cases, and this generally requires a small amount of time compared to the scoring. Moreover, if the number of changes in the software is small, test cases which do not cover any change are more likely, and thus the zero equivalence class is larger. As these tests can be just concatenated with the partial permutation produced when all test cases from other equivalence classes have already been arranged, the QS evaluation can be further sped up.

Considering time as the constrained resource in the software testing phase, the EET_r results for the all techniques is presented in Figure 4.18. EET_r values shows the effective times of the techniques, aggregated by project. Each bar a plot shows EET_r of a specific technique on a particular project. The bars are related to the *average* effective times between version pairs and variants in each project, and the error bars show the standard deviation. Each bar highlights the time employed by the application of a specific technique, in orange, and by the subsequent execution of the minimum fraction of test cases in the permutation which discovers all faults, colored in blue. According to what stated for the technique execution time of ART and Genetic-APTC, these approaches have been

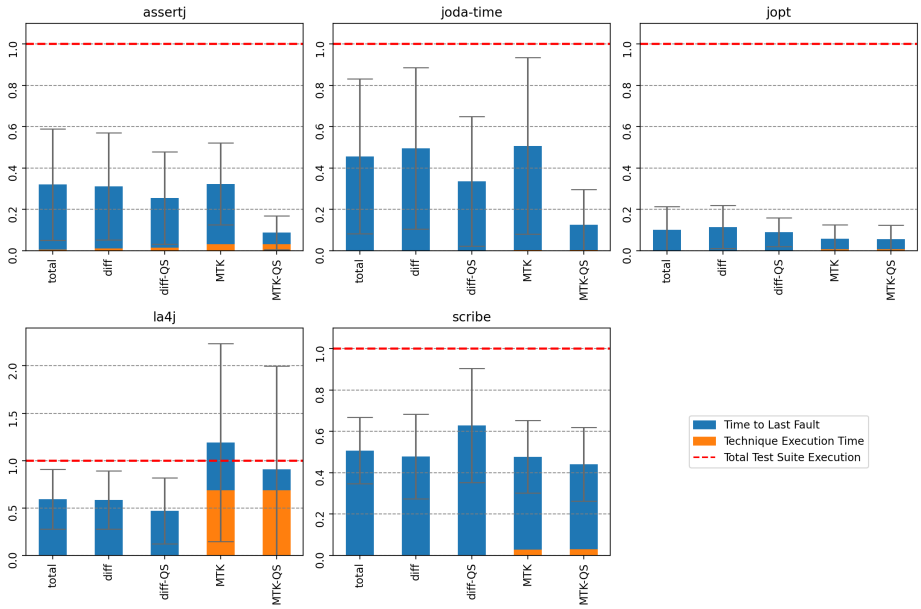


Figure 4.18. Average EET_r values between projects for MTK, MTK-QS and the baseline techniques.

excluded from the analysis again. In fact, their execution time alone exceeded the whole test suite execution time in all cases.

As it can be seen from Figure 4.18, the proposed techniques perform worse than the baselines on *La4J* projects. Analyzing only the slice of the test cases needed to be executed, MTK produces comparable results with respect to the baselines, while MTK-QS has a lower and better score. However, when the execution time of the technique is also considered, the EET_r of MTK tends to be greater than 1, and thus the whole process of finding a permutation and executing test cases is longer than the time to execute the whole test suite. MTK-QS on this project provides EET_r values slightly lower than 1 on average, thus allowing to spare time with respect to executing the whole test suite. However, EET_r results of other baselines are lower and allow to save a greater amount of time. It is also possible to see that the standard deviation of the proposed techniques is very high, due to the high variance of MTK and MTK-QS execution time

on the various versions of *La4J*. Indeed, when large methods in the project have not been changed between the two versions, the evaluation of PTK is not performed on them, and thus the execution time is significantly shorter. As *La4J* presents pairs of versions in which large-sized methods have not been modified, the techniques produce lower EET_r results. When aggregated on a project basis, however, the elapsed times significantly differ and the variance in the obtained values is increased. For these reasons, however, MTK and MTK-QS on *La4J* do not produce any benefit in the effective time of the testing phase with respect to the baselines.

For *Scribe* and *JOpt* projects, it is possible to see that both MTK and MTK-QS produce results which are comparable to the baselines. In both cases, the technique execution time is negligible compared to the slice of the test suite needed to discover all faults. In particular, the MTK-QS approach seems to produce, on average, lower EET_r values, constructing permutations which provide more benefits to the testing activity.

On *AssertJ* and *Joda-Time* projects, the time to build the permutation of MTK and MTK-QS is still negligible, accounting to around 3% of the total test suite time for *AssertJ*, while being under the 1% of the suite time of *Joda-Time*. These projects are the largest in size and with the highest number of test cases among all subjects in the collected dataset, but construction of the permutation introduce small overheads in the testing phase. Thus, their EET_r results on these projects are dominated by the fraction of test cases in the permutation which needs to be executed to uncover all faults. Focusing on MTK, its EET_r values are in general worse or at most comparable to the baselines, due to its redundancy issues. This leads again to poor performances in comparison with all other techniques. On the other hand, the EET_r results for MTK-QS are significantly lower than the baselines on these two projects. In particular, the second best-performing technique, i.e., diff-QS, produces effective execution times which are at least doubled than the EET_r values of MTK-QS. This suggests that the application of MTK-QS can remarkably reduce the testing efforts for these projects.

To assess the statistical significance of whether EET_r produced by MTK and MTK-QS were greater or lesser than the baselines, a Mann-Whitney-Wilcoxon U-test analysis has been performed. Its results are resumed in Figure 4.19. Green cells in the figure means that the null

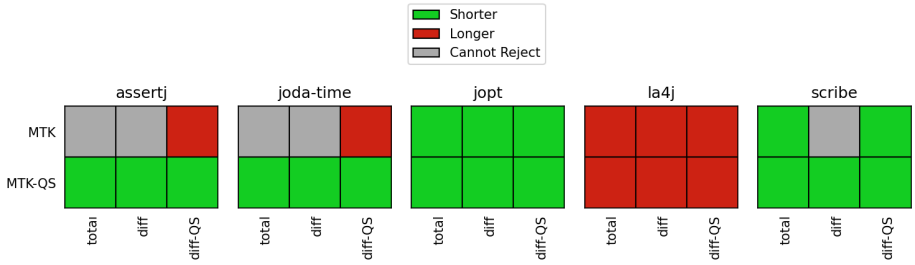


Figure 4.19. Mann-Whitney-Wilcoxon U-Test comparing effective execution times of techniques. Cell colors represent whether the technique in the row generates distributions with shorter (green) or longer (red) effective execution times than the technique in the column. Cells in gray indicates that the EET_r distributions do not differ in a statistically significant way.

hypothesis $\mathbf{L_0^{t_1,t_2,p}}$ in Definition 4.11 can be rejected with high degree of confidence, and thus that the technique t_1 presented on the row has significantly less effective times than technique t_2 on the respective column, for a particular project p. Conversely, red cells means that the null hypotheses $\mathbf{G_0^{t_1,t_2,p}}$, presented in Definition 4.11 as well, can be confidently rejected, and EET_r values of t_1, on the row, are significantly greater than those produced by the technique t_2, on the column, for project p. When it was not possible to state the significance difference of the results, a gray cell has been drawn.

For the *AssertJ* and *JOpt* projects, the MTK approach produces EET_r results which are neither lower or greater than the total and diff baselines. On the other hand, MTK has longer effective execution time than the diff-QS approach, and tends to discover all faults in the software significantly slower than the latter approach. On *JOpt* project, MTK produces statistically significant lower EET_r values than the baselines, rejecting the null hypothesis $\mathbf{L_0^{t_1,t_2,p}}$ in all cases. However, all techniques on this project tend to exhibit low EET_r results, and even if MTK performs significantly better, in general all permutations produced by any considered technique allow to save testing time. A similar scenario arise from *Scribe*, where the MTK approach is better than total and diff-QS baselines, while producing the same distribution of EET_r values with respect to diff. In this latter case, neither $\mathbf{L_0^{t_1,t_2,p}}$ nor $\mathbf{G_0^{t_1,t_2,p}}$ null hypothesis can be confidently re-

jected. Finally, MTK performs statistically worse than all baselines on the *La4J* project, as expected. This is due to the fact that the application of MTK alone to create the permutation consumes more time than the execution of all test cases in the suite.

MTK-QS approach produces significantly shorter effective execution times with a very high level of confidence almost all projects, and it can be used to save more time in the testing phase than the other approaches. The same observations on MTK applies for *JOpt* and *Scribe*, i.e. that even if the technique produces permutations with statistically lower effective time, any other technique has similar EET_r values, thus the amount of time spared by different techniques is similar. Instead, when MTK-QS is applied to *AssertJ* and *JOpt*, it produces significantly better permutations in terms of EET_r, allowing to reduce the time needed for the software validation phase with very high confidence. Focusing on *La4J*, it is possible to see that MTK-QS has significantly larger values of EET_r and the null hypothesis $\mathbf{G_0^{t_1,t_2,p}}$ is confidently rejected. This is in line to what observed on MTK, as, regardless of the quality of the produced permutation, the evaluation of the PTK function on this project depletes a vast amount of time resources allocated for the testing phase, lasting almost as the execution of the entire suite.

In general, the EET_r performance of MTK in terms of effective times was satisfying on the smaller projects (*JOpt* and *Scribe*), while it produced results which were worse or at most similar to the baselines on the largest projects. Conversely, MTK-QS outperforms the other techniques in almost all scenarios, accounting for notable benefits for the testing phase. MTK-QS performed particularly well on the largest projects, with great size and a vast number of test cases. This suggests that the application of MTK-QS on large projects can produce high benefits and significantly reduces the resources allocated for the regression testing phase of the software. However, this observation cannot be generalized to all types of software. MTK-QS, as well as MTK, have been definitely proven unsuitable for the *La4J* project. Based on the in-depth execution time analysis on this project, it can be noted that exceedingly long methods, with hundreds of lines of code and long sequence of operations at the same nesting level, are the main weakness of MTK and MTK-QS. However, very long methods are not advised and avoided in modern developmental practices, and

thus this weakness can be seen only in specific and niche kinds of projects. For these projects, it is advisable the usage of different RTP techniques, as even if permutations produced by Tree Kernel-based prioritization can have high fault-detection performances, the cost to create these permutations could be not affordable when strict time constraints are imposed to regression testing activities.

4.5.3 Threats to validity

The empirical evaluation performed on MTK and MTK-QS techniques may be conditioned by threats affecting its validity. These threats could have affected the results and the findings, possibly limiting the generalizability of performed experiments. As done in the empirical evaluation presented in Section 3.3, to decrease the chance of misinterpretations, several common and widely-used guidelines were employed while designing the study [140]. However, some threats may be still present, and subsequently analyzed.

Threats to internal validity

The confidence of association between treatment conditions and the establishment of the results can be limited by confounding variables, to which internal threats to validity refers.

One threat is related to the implementation of MTK and MTK-QS approaches, due to errors and bugs in the source code designed to implement the techniques. To reduce the possibility of bugs in the used implementation, the module implementing the AST generation relies on the *GumTreeDiff* library. This is a library which has been used in several sftware engineering studies related to the analysis of tree-based representation of source code. To evaluate the PTK function used in both MTK and MTK-QS, the robust implementation of Tree Kernel function provided by the *KeLP* framework has been employed as the foundation of the proposed RTP technique. Both *GumTreeDiff* and *KeLP* have been widely used in literature, and are open-sourced and well-maintained by their respective communities, reducing the risk of unexpected behaviour in their execution.

Another threat is related to the coverage reports used by the evaluated RTP techniques. The collection of these reports has been performed by the

OpenClover tools, which is widely used in many research and industrial contexts to obtain test coverage for Java software. The module used to provide per-test coverage has been written from scratch, extending the base functionalities of the *OpenClover* tool. However, the per-test results were already evaluated by *OpenClover*, and the performed modifications were applied only in the module which produced the *XML* reports. Reports produced by the tools are used by all MTK and MTK-QS, along with the baselines, which thus leverage the same coverage information.

According to the baselines, the implementation of total, ART and Genetic-APTC used in the empirical evaluation is the same employed in different RTP studies [82, 83], with the addition of some *wrapper* code to integrate these techniques in the common prioritization pipeline on which MTK and MTK-QS rely. The diff prioritization was entirely implemented, using the *git-diff* utility to extract information on changed methods between the two versions. As this utility is consolidated and widely-used by a vast number of software engineering practitioners and scholars, it is robust and well-maintained, and its usage limits the chance of mismatch when evaluating changes.

To reduce differences between MTK-QS and diff-QS when re-arranging test cases according to the quotient-set, an higher-level module implementing the general QS approach has been designed, allowing to inject the specific score function via a *Dependency Injection* pattern [47]. The implementation of MTK-QS and diff-QS has been performed using the same QS implementation, which receives directly the test suite whose test cases were labeled with the scores assessed by the respective techniques. This limits any discrepancy between between MTK-QS and diff-QS when the common last step of re-arranging test cases needs to be performed.

Concerning the internal threats to collecting the execution time of the various techniques, the JMH framework has been employed. This framework reduces the chance of noise in the evaluation of benchmarks, providing more reliable results. It is however possible that uncontrollable external factors could gave rise to fluctuation in the registered times, for example due to JVM optimizations and to the operating system routines of the machine in which the experiments were ran. To limit these undesired effects and ensure fairer results, several best practices proposed in literature [33] were adopted while the benchmark code has been written. The EET_r anal-

ysis included both the execution time of the various employed techniques, and the duration of each test case executed in the suite. For these latter times, the test execution reports provided by the *Surefire* tool have been used. Surefire automatically reports the time consumed by each test case in the test suite, when executing the validation phase. To guarantee the consistence of test execution time with technique execution time, the validation phase has been executed on the same machine used to collect the latter times, for all pairs of software versions. The resolution of execution time reported by the Surefire tool is 1 millisecond, thus a time of 0 has been reported to the test cases which employed less time. For this reason, it is possible that the EET_r included in the study were lower than the real ones. However, according to an in-depth analysis of the test execution of all projects, this phenomenon seldom occurred in the dataset, and absence of execution time for those tests had a small impact on the total time of a test suite. Anyway, the normalization of the execution times over the entire suite time reduced the extent of these discrepancies.

An additional threat to the validity of our conclusions and statistical inferences is posed by the *multiple comparisons problem*. This problem cause an increase in the likelihood of false-positives to appear, as the more comparisons are performed, the higher the probability that significant results appear by chance, when testing a family of hypotheses. This can lead to the erroneous identification of significant effects or relationships in the treatment. To mitigate this threat, it is a common practice in software engineering studies to adopt a Bonferroni correction [118, 92]. This correction helps to adjust the threshold significance for the individual tests, and ensures that the error rate for the family of comparisons is still controlled.

Threats to external validity

Threats to external validity may limit the generalization of findings in a study. To limit the sampling bias, the empirical evaluation of MTK and MTK-QS involved a subset of five open-source Java projects belonging to a broader set, whose projects have been used in different RTP studies [82, 83]. To obtain the subset on which the empirical evaluation has been performed, the projects have been selected to be as heterogeneous as possible in their purpose, size, number of test cases, and extent of changes between various project versions. However, this limited number of projects might

not be indicative of all possible Java projects.

Another threat to external validity is the usage of injected faults through software mutation, as real faults have not been available in subject projects. This approach has been employed in many works on the prioritization topic [82, 83] and can be appropriate for research when the environment is controlled [70]. Nevertheless, mutation faults might not be representative of real faults in some contexts [106]. To mitigate this, the faults have been injected only in methods which were already changed between the software versions considered in each experiment. In this case, the injected faults should resemble mistakes made by developers when rewriting methods. Furthermore, to diversify as much as possible the types of faults, all the mutation operators provided by the *Major* tool, shown in Table 3.3, have been activated.

Chapter 5

A Dataset of Software Projects with Real Faults

Few people have the imagination for reality

Johann Wolfgang Von Goethe

RTP approaches proposed in literature typically involve an empirical evaluation of performances on a dataset of benchmark projects, and a comparison of such performances with well-known state-of-art baselines. As prioritization techniques essentially seek for a permutation with the desired property of high fault-detection rate in a large search-space, the experimentation has a major role in assessing the effectiveness of RTP strategies. However, the collection of a benchmark dataset is usually non-trivial, for two main reasons: the heterogeneity of information leveraged by various prioritization approaches, and the typically lack of faults in open-source public projects, which obliged researchers to use synthetic faults in their experimentation.

To this end, part of my research aimed to design a dataset which could be straightforwardly applied to RTP experimentation, allowing scholars and practitioners in the prioritization field to focus on the design and the experimentation of techniques they propose, rather than on the collection of benchmark subjects. This research resulted in *ReCover*, a dataset of software projects which has been collected through mining online software

repositories and whose projects include real-world faults. It has been proposed in my paper [6], and consists of 28 open-source public Java projects with 228 different versions across these projects, subdivided in 114 software evolutionary scenarios.

This chapter presents the *ReCover* dataset and some of its application. The first section describes the motivations behind the design of *ReCover*, along with the procedure which has been followed to mine remote software repositories in order to obtain the projects and versions within. Furthermore, the section details the structure of the dataset and presents some statistics and characteristics of the collected projects. The second section introduces two applications of *ReCover*: a study on fault-proneness metrics based on *Artificial Intelligence* models, presented in the paper [7], and an experimentation of various RTP techniques on the dataset.

5.1 The ReCover Dataset

This section presents the *ReCover* dataset, providing motivations behind its collection and detailing the structure of the tool designed to perform the mining process starting from two popular RTP datasets, and the steps to enrich data in these datasets. Subsequently, the section focuses on the *ReCover* dataset which has been obtained by the collection process, and provides details and insights on the collected software versions.

5.1.1 Why a Dataset for RTP?

Experimentation in the field of RTP is heavily empirical, and typically involves three steps: i) the execution of different prioritization strategies on a set of benchmark projects, then ii) the evaluation of some desired metrics, usually related to fault-detection rate, on produced permutations, and iii) eventually the comparison of results with different chosen baselines. Nonetheless, these steps presuppose the availability of a consolidated set of benchmark projects, which is seldom easily obtainable. Indeed, there is no generally accepted shared set of benchmark projects on which experimentation can be performed, and researchers should typically collect the needed data by themselves. Moreover, the software projects collected to experiment a RTP technique rarely include auxiliary information which

are not directly used by the specific technique in evaluation, aggravating the step of performance comparison by other novel proposed approaches.

For the sake of experimentation, subjects of an empirical evaluation in RTP should resemble software evolutionary scenarios. Each scenario is typically composed by a pair of software versions: the version before the evolution (i.e., the *previous* version) and the *current* evolved version under-test. As fault-detection rate is usually the property which a prioritization should maximize, the current version in a pair should include at least one *regression fault* which is discovered by test cases in the suite.

Usually, finding software versions including real-world faults exhibited by test cases is not a trivial task. In fact, in evolutionary scenarios of open-source projects it is typically mandatory for developers to execute the testing phase locally, before submitting their modifications to the codebase. However, the submitting process could not be allowed if the verification phase has not successfully executed without any test failures. If it is not the case, developers are required to fix the failures and to repeat the testing phase, until all test cases pass. Only in this case they are allowed to eventually submit the updated version. For this reason, open-source projects rarely present software regressions and cannot be employed in RTP experimentation without further processes aimed to add faults in the updated version.

This reason leads the vast majority of RTP evaluations on open-source projects to leverage synthetic faults, which can be either manually or automatically injected to resemble defects in the software [40, 82, 83]. This kind of experimentation frames a *controlled environment* in which execute the evaluation of techniques, maximizing the information on the injected faults. In fact, the nature and the location of each fault is known, and this knowledge allows an accurate evaluation of fault-based performance metrics.

Traditionally, some datasets have been designed to be employed in software engineering research, such as *SIR* [39]. This repository includes software projects in different programming languages, each with various versions, and which have been used in several fields of software engineering, including RTP. *SIR* provides the full source-code of both toy experimental projects and real-world applications, which however do not present real-world faults. To cope with the issue, synthetic faults have thus been

manually injected into the repository by software engineering practitioners, to resemble real defects. Nonetheless, projects written in object-oriented programming languages which are included in *SIR*, are often outdated. Focusing on Java-based projects, several of them use older versions of the language grammar. On one hand, this causes the projects to be not fully representative of modern software development, and on the other hand increases the efforts of building and testing the projects to obtain auxiliary information, such as coverage reports, due to the deprecation of previous language structures.

More recent datasets leverage open-source software projects with artificial faults, automatically injected via code mutations. The dataset used as the basis of the experimental evaluation of Chapter 3 and Chapter 4 [82, 83] are part of this category. These datasets typically included the full source code of projects and versions contained within, along with the location in which faults have been injected. They also provide coverage information, usually in the form of *coverage matrices*[1], at statement, branch and method levels of granularity. However, this kind of coverage information could limit prioritization approaches jointly based on coverage and code churn, as these datasets typically lack of mapping between the covered structural elements and column indexes. For this reason, it might be necessary to rebuild these software in order to collect more detailed information.

One of the main limitation for dataset including synthetic faults is the threat to the generalizability of the performed RTP studies. Although different studies state a high correlation between artificial and real faults in many contexts [40, 70], synthetic faults might not be representative of all possible scenarios of software defects, thus limiting the validity of experiments if the injection process is not accurately performed [51].

Differently from open-source projects, industrial systems often include information of test failures during their evolution, and could be a valuable benchmark for RTP techniques due reported real-world faults. Indeed, the testing process of these systems is often performed in the company infras-

[1] A *coverage matrix* is a Boolean matrix $M \in \{0,1\}^{n \times m}$, where n is the number of test cases and m is the number of code elements in the software. The entry $M_{i,j}$ is equal to 1 if the test case labeled with i covers the code unit j, otherwise it is 0. Code elements can be considered at different levels of granularity.

tructure, and feedbacks on failing tests are readily available to development teams in order to correct these defects. Some RTP studies evaluate the techniques they propose on such industrial systems [27, 36, 43, 86], and the validity of results is strengthened by the application to these practical cases. However, industrial systems are typically proprietary applications, and their source code cannot be shared to replicate the experiments. Furthermore, some RTP techniques are developed specifically for these systems, tending to adhere to the company developmental methodology. For this reason, it could not be completely clear how they could perform in different contexts.

Lately, the spread of public code repositories and CI/CD pipelines, such as *GitHub*[2] and *TravisCI*[3], led several developers to migrate the execution of the testing phase of their software to these remote environments. Along with the execution of the build and testing phase, CI/CD pipelines often publicly store test execution reports, and it is thus possible to query these reports to detect real-world faults in the project versions.

The presence of public test execution reports allowed the mining of open-source projects from online code repositories. This led to the collection of various datasets to employ in different tasks of software engineering, such as *GHTorrent* [52], *TravisTorrent* [17] and *Defects4J* [69]. These datasets record a set of metrics observing several events which occur to projects and versions stored in the repositories, e.g., a new commit or the completion of lifecycle phases in a pipeline. However, constructing a dataset for RTP research from the recorded events is not straightforward, as shards of information from different sources should be aggregated together to retrieve suitable evolutionary scenarios to employ in empirical evaluations.

The recent *RTPTorrent* dataset [88] has been specifically designed to support prioritization researches, as its name suggests. *RTPTorrent* contains references of over 100k software evolutionary scenarios extracted from *GHTorrent* and *TravisTorrent*. Another large dataset for RTP has been used in the empirical evaluation of a *Information Retrieval*-based RTP technique [107]. The study employed a dataset with over 3k scenarios of software evolution with real regression faults, and presented references to

[2] https://github.com/, visited on 25/01/2024.
[3] https://www.travis-ci.com/, visited on 25/01/2024.

the location of each software version in its remote repository. Both the former and the latter datasets include real-world faults.

Reference-based datasets are useful when experimenting particular types of RTP techniques, such as history-based prioritization approaches, but might be limited when techniques employing other types of information should be experimented. In these scenarios, researchers need to manually retrieve the original version of the software from the repository and possibly re-execute the building and testing phases to collect auxiliary information (e.g., code coverage or test execution reports), which is a time-consuming task. Furthermore, in several cases it is not possible to recreate the original build environment, causing the failure of the collection process, or the remote repository pointed by the references might have been moved or deleted, making references not stable in time.

These limitations of the several datasets proposed for RTP motivated the process of collection of a novel dataset which could alleviate the burden of collecting, configuring and possibly executing the projects on which evaluate the effectiveness of prioritization techniques, as a step towards an easier empirical replication.

5.1.2 Mining ReCover

To collect a dataset through public software repositories, the first step has been to outline projects which could be suitably applied to RTP studies. To this end, a literature analysis outlined two of the most complete and recent sources: *RTPTorrent* [88], as it is specifically tailored for RTP applications, and the dataset in [107], which will be referred to as *IR dataset* for the remainder of the chapter.

RTPTorrent includes a heterogeneous collection of open-source Java projects with the Maven project management tool. The projects have been jointly mined from *GitHub* and *TravisCI*. To collect *RTPTorrent*, several build logs available on *TravisCI* have been automatically analyzed and those builds which exhibited test failures were considered. On the *TravisCI* platform, build logs also record the identifier of the *GitHub* commit which triggered the build, hence allowing the extraction of the two versions in the evolution of the project from these failing builds. More specifically, the version related to the commit triggering the build is designed to be the *current* version under test, while the commit immediately preceding it is

designed to be the *previous* version. Pairs of versions identified by builds in this manner are included in *RTPTorrent*, which counts over 100k pairs across 20 Java projects. The builds are stored in CSV format, including the build identifier on *TravisCI*, the commit SHA identifier of both versions on *GitHub* and other aggregated data as the number of test cases, failures and the test suite execution time on the *TravisCI* platform. *RTPTorrent* has been chosen as its projects presented a good level of maturity and popularity on *GitHub*, and are heterogeneous both in nature and in size. On the other hand, the collected builds are not always characterized by code changes, as it is possible that the same version identifier is used in several builds.

The *IR dataset* contains several software projects, along with different builds related to evolutionary scenarios. On this dataset, the empirical evaluation of an *Information Retrieval*-based technique has been performed [107]. The dataset has been similarly obtained as *RTPTorrent*, by mining *TravisCI* and *GitHub*, and contains projects which use *Maven* as project management tool. The methodology for retrieving the dataset consisted in initially identifying two consecutive *TravisCI* builds, for which the older is successful and the newer is failing due to test case failures. From these two builds, the identifiers of commits triggering them have been then extracted and the versions were retrieved from *GitHub*. The dataset contains over 2000 builds across 123 Java projects, storing information of each build and commit identifiers in CSV format. The main advantage of *IR Dataset* is the fact that collected pairs of versions are more likely to exhibit failures due to changes in their source code, due to have been employed on a change-aware RTP approach.

RTPTorrent and *IR Dataset* provide data about projects and versions through references to the remote location of builds and commits, respectively linked to *TravisCI* and *RTPTorrent*. To enrich the data with different information which can be readily used in RTP studies, the full source code and test coverage reports needed to be collected. Due to the large number of builds and projects jointly contained in both *RTPTorrent* and *IR Dataset*, a manual retrieval of this data would be unfeasible, as the process involves both a re-build of every single version, enriched with code coverage tools if not already present in the original project, and the execution of the test phase. Thus, this process has been automatized through

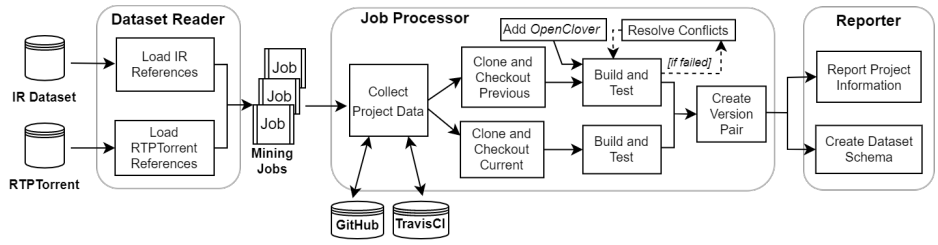


Figure 5.1. Steps and operations performed by the mining tool used to collect the *ReCover* dataset.

the design of a mining tools which has been used to collect the *ReCover* dataset.

Figure 5.1 shows the pipeline of the designed mining tool[4]. The tool is written in Java and is capable to query both *TravisCI* and *GitHub* to retrieve versions, download their source code, and to execute all their building and testing phases to obtain code coverage and test reports. The overall architecture of the tool is divided in three sequential modules: the *Dataset Reader*, the *Job Processor* and the *Reporter*.

The Dataset Reader is responsible to process the build information of *RTPTorrent* and *IR Dataset*, in their native CSV format, and to extract the list of *mining jobs* which should be subsequently retrieved. The Dataset Reader module does not actually query any remote software repositories, as this will be done in following steps, to reduce the latency between the identification of mining jobs and their effective retrieval process. Each mining job is identified by a *slugname*, i.e., a string with the name of authors and the name of projects concatenated by a '@' character, the *build identifier* to locate the specific build on *TravisCI*, and two commits SHA identifier for the *previous* and the *current* versions involved into the build. Commit identifiers can be empty, whether these were present in the original dataset or not. The module is also responsible to remove duplicate entries from the initial datasets. To this purpose, two builds are considered duplicates if they shares the same commit identifiers of both previous and current versions, as they are indiscernible from each others. As a note, the Dataset Reader module has been implemented to readily

[4]The complete source code of the tool is available at https://zenodo.org/records/5911108.

support different input datasets by adding new sub-modules capable of producing the related list of mining jobs.

After the list of mining jobs has been collected by the Dataset Reader, the Job Processor is designed to retrieve the data of each build and to execute the lifecycle phases needed to collect the coverage and test information. Each mining job is processed in parallel by the Job Processor module, as the steps of different jobs can be performed independently, to significantly speed up the collection of the dataset. Starting from a mining job, the Job Processor queries the data on *GitHub* and *TravisCI* according to the references contained in the entry. In particular, if the commit SHA identifiers for both versions are present, *GitHub* is queried directly to obtain the original sources of the versions. If any identifiers for previous or current version is missing, then the module tries to retrieve this information by querying *TravisCI* via the build id of the mining job. If the information could not be found, e.g., because the storage time of the build expired on *TravisCI* platform, or the identifiers are no more valid references on *GitHub*, the entire job is marked as failed due to references not found and the process ends after communicating this status to the Reporter module.

When the *GitHub* references have been found for the job, the related repository is cloned and checked out to the specific previous and current versions, which are then stored in separate folders to start the building process. The build and test phases of the versions is performed through the Maven tool, by running the related lifecycle goals. While the current version can be built as it is, the previous version needs to be built to allow the record of coverage reports. To this end, the *OpenClover* coverage tool is added to the dependencies of the previous version, before starting the build phase. If there are conflicts with this dependency, the tool automatically tries to resolve them by checking and removing other known dependencies which could clash with *OpenClover* (e.g., other coverage tools such as *JaCoCo*[5]). If any build phases result unsuccessful, either for previous and current version, the job is marked as failing and the status is communicated to the Reporter module.

After the building phases have been completed for both versions, their test phases is subsequently invoked. The test phase on the current version

[5] https://www.eclemma.org/jacoco/, visited on 22/01/2024.

allows to obtain test reports and thus information on failing tests, while the testing phase on the previous version is needed to retrieve coverage reports. These reports are produced at statement granularity level, as *OpenClover* automatically includes also the method and branch level granularity. The entire build is marked as failed if the current version does not show any test case failures. Note that if any test cases fails in the previous version, the build could be still acceptable and it is further processed. If all these steps have been successfully completed, the pair of versions is inserted in the dataset, and the Reporter module is informed of the success.

The Reporter module is responsible to produce the meta-information related to the collected version pairs and to aggregate metrics concerning the mining process. When the Job Processor communicates the status of each job, either success or failure due to missing references or errors in build and testing phases, the Reporter module internally aggregates these data. After the Job Processor finishes the collecting process for all mining jobs, the Reporter module produces two types of different reports: a relational structured format, which is stored in a *SQLite*[6] database and can be queried using SQL, and a XML textual format. The first report contains details on projects, builds and versions, along with general statistics of the mining process. The latter report includes specific information of projects and builds, such as the size of each mined version, the number of common test cases in a version pair and the percentage of covered code units in the previous version of each pair.

At the end of the execution of the entire mining process, several builds were failing, meaning that the mining could not be performed. Concerning *IR Dataset*, from the over 2000 builds present in the dataset, less than an half were found on *GitHub* and produced a *reference not found* status. Furthermore, another significant fraction of the builds resulted in an error status, often due to errors in the building phase of its versions, and seldom because the current version exhibited no failure in the testing phase. In summary, less than 10% of the original *IR Dataset* could have been mined successfully. Figure 5.2 shows the percentage of build in *IR Dataset* in each status, and the number of collected version pairs. In particular, from this dataset 33 projects and 124 version pairs have been collected.

When analyzing references from *RTPTorrent*, less than 3% of the over

[6] https://www.sqlite.org/index.html, visited on 21/01/2024.

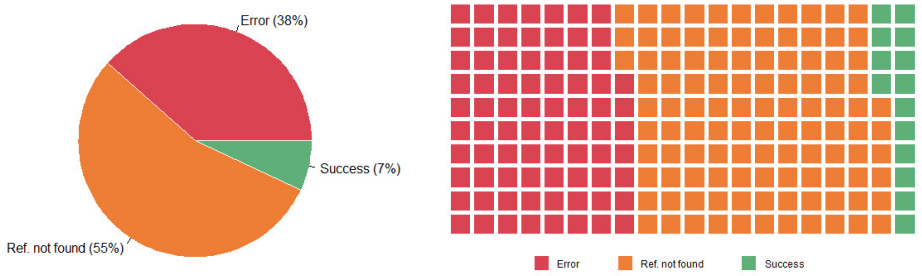


Figure 5.2. Results of the *ReCover* mining process on *IR Dataset*. Left: percentages of processed jobs according to their status. Right: absolute number of jobs per status (10 jobs per block).

100000 builds in the dataset have been found. The main cause of this issue is due the expiration of build logs on *TravisCI*, which could not be successfully queried to obtain the SHA identifier of previous versions. Among found references, only 78 pairs of versions for 5 different projects have been successfully collected[7].

From the total of 202 pairs jointly collected from *IR Dataset* and *RTP-Torrent*, a manual inspection led to the removal of 88 different pairs, due to test failures ascribable to configuration issues (e.g., missing environments and unavailable remote service access). The removal of these pairs had also the effect to the exclusion of 10 projects, which had no version pair left. For this reason, the collected pairs included in *ReCover* are 114, spread across 28 projects.

5.1.3 Dataset Insights

ReCover contains 28 open-source Java projects and 228 software versions divided in 114 pairs of evolutionary scenarios. Table 5.1 presents the complete list of projects collected in the dataset, along with some software metrics as the number of builds collected for a specific project, its average number of LoC and methods, the average percentage of statements covered by tests of the previous version in a build, the number of test cases and

[7]No figure has been inserted for the mining process of *RTPTorrent* references, as it would have been resulted in an almost full orange-coloured pie-chart.

the average faults across pairs in the project. As it is possible to see, the number of builds for each project is typically small (17 projects contains 1 or 2 version pairs), while other projects have a high number of collected builds, up to 22. The collected projects are heterogeneous and vary in their size, from tiny projects with less than 3k LoC, to software projects with more than 100k LoC and over 5k methods. Also the percentage of coverage among the various builds highly varies, and could be used to experiment the effects of different amount of coverage information in RTP techniques.

Project	Builds	LoC	Methods	Covered	Tests	Faults
abel533@Mapper	2	7k	1598	56.5%	226	4
adamfisk@LittleProxy	21	2k	465	79.8%	61	7
alexxiyang@shiro-redis	2	<1k	164	49.0%	20	9
alibaba@fastjson	3	31k	15141	50.1%	4082	11
alipay@sofa-rpc	4	25k	4498	70.6%	484	46
apache@incubator-dubbo	4	50k	10598	63.8%	2563	36
apache@rocketmq	5	56k	6960	43.6%	310	5
AxonFramework@AxonFramework	22	34k	8872	82.4%	1678	11
ctripcorp@apollo	1	19k	3490	68.5%	574	3
davidmoten@rxjava-extras	2	6k	1448	72.3%	631	11
demoiselle@framework	3	2k	626	70.7%	196	5
dynjs@dynjs	2	21k	3436	80.9%	852	20
elasticjob@elastic-job-lite	1	6k	1771	85.1%	193	11
hs-web@hsweb-framework	1	16k	3278	27.1%	23	5
JSQLParser@JSqlParser	1	9k	2602	25.3%	744	1
l0rdn1kk0n@wicket-bootstrap	1	16k	2995	28.8%	205	1
lukas-krecan@JsonUnit	2	4k	911	92.9%	1861	28
magefree@mage	2	182k	115382	0.7%	2331	2166
mitreid-connect@OpenID	1	18k	2057	43.9%	220	2
onelogin@java-saml	1	3k	749	96.1%	359	25
pf4j@pf4j	3	3k	480	74.8%	60	2
pippo-java@pippo	1	11k	1638	26.3%	136	1
rapidoid@rapidoid	11	44k	7487	53.1%	492	106
rickfast@consul-client	1	2k	418	49.5%	50	1
sismics@reader	7	8k	1083	58.0%	94	19
tcurdt@jdeb	1	3k	443	76.5%	77	1
thinkaurelius@titan	8	19k	2603	62.5%	263	15
vipshop@vjtools	1	7k	1398	67.4%	170	7

Table 5.1. Overview of projects included in *ReCover* and their statistics.

Figure 5.3 presents box-plots of some statistics of versions contained in *ReCover*. The plots show the distribution of versions according to three metrics: the number of LoC, which is an indicator of the size of the collected versions, the size of the test suite, which gives insights on how thoroughly the software has been tested, and the number of failing test cases, which relates to the defects present into the current version of each

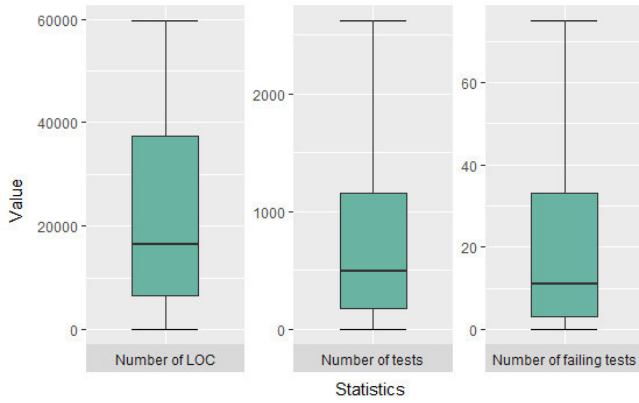


Figure 5.3. Box-plots presenting some statistics of projects and builds in the *ReCover* dataset.

pair. Outliers have been excluded from the figure. According to the size, a significant number of versions are medium and medium-large sized, and distribute around 8k and 38k LoC, with a median value of 18k. *ReCover* presents also small projects, with versions less than 7k LoC, and large projects, with more than 50k LoC. From the perspective of number of test cases, the median cardinality of the test suite is around 500 tests, but a significant number of versions fall in the range 200-1200 test cases. Furthermore, a small number of versions possesses more than 1200 test cases. The execution time of the test suites also significantly varies between projects, ranging from times of less than a minute to projects whose execution time is larger than a hour. If contextualized in CI/CD environments, even project with short execution times of their test suites, when framed in CI/CD contexts, might be suitable subjects for RTP. In fact, as the frequency of updates grows larger, the time window to execute test cases becomes narrower and, in extreme cases, even short overall execution times might not be affordable. Concerning the number of failing tests between pairs of versions, a high amount of pairs presents from 5 to over 30 failures and, in some pairs, they exceed 40 failures. These statistics show a reasonable amount of heterogeneity among the collected versions, and guarantee a good level of generalizability when used as basis for the experimentation of RTP techniques.

Figure 5.4. *ReCover* structure and report formats. (a) Structure of folders for projects in *ReCover*. (b) UML class diagram for the database schema. (c) Snippet of project meta-data in XML reports.

Project versions in *ReCover* presents a particular structure of folders in the repository, as Figure 5.4a shows. Each mined project is stored in a folder whose name corresponds to the *slugname* of the project. Insider the folder, each pair of versions is stored in a sub-folder named after the build identifier in *TravisCI*, and its pairs of versions are stored in two nested sub-folders with the *GitHub* SHA identifiers. As the alphabetic order of version identifiers does not resemble their chronological order, a text file defining which version is the previous one and which is the current one is present in each build folder. Each version folder contains the full original source code of the commit, along with surefire reports and, in case of previous versions, the per-test coverage reports produced by *OpenClover*.

Meta-data about projects and versions collected in *ReCover* are produced by the Reporter module of the mining tool in relational and XML format. Figure 5.4b depicts the class diagram with the structure of *SQLite* database created by the Reporter module after the mining process. Each project possesses its name and its source dataset (either *RTPTorrent* and *IR Dataset*), and it is composed by one or more version pairs, which constitute the evolutionary scenarios and are related to a specific build in *TravisCI*. Each version pair is linked to a previous and a current version. Versions stores the SHA identifier of the *GitHub* commit, the number of LoC and classes, and the timestamp of the commit. Furthermore, test cases of each versions are included in the database and are associated to a specific version in *ReCover*. Each test case is characterized by its full qualified name and the status, either *passed*, *failed* or *error*. If the test has been successful in the testing phase of the mining process, then its status

is "*passed*", otherwise it is "*failed*" if the test did not succeed because of a failed assertion, or "*error*" if other unexpected causes took place (e.g., an exception has been thrown during the test execution). A test cases is considered as fault-exhibiting if its status is either *failed* or *error*. The database has been designed to readily query the data, in particular related to test cases, in order to help *ReCover* users in aggregating project statistics.

The produced XML reports are thought to be a catalog of projects and versions in the dataset. By reading the reports, it is possible to find various information on the structure of versions. Figure 5.4c shows a snippet of the XML report related to a specific project. The `artifactData` element stores the information on the entire project in *ReCover*, as name and author, along with the type of project. A project type is either "*modular*", if it aggregates different sub-modules with their own separated source files, or "*plain*". Useful metrics of each project are stored in the `artifactMetrics` element, as the number of version pairs included or the average LoC.

Version pairs are characterized by the *TravisCI* build id and specific metrics for the pair, such as the modifications to the test suite which have been done in the evolution between the two versions. In particular, attributes `addedTests` and `deletedTests` show the number of tests which have been respectively added and deleted between the versions, while `commonTests` records how many test cases remained unchanged. Additionally, the attribute `failures` counts the number of failing test cases between the versions. Previous version and current version have two different tags, to readily find their role in the pair. Both elements have the same attributes, i.e., some version metrics and the percentage of statements covered by test cases (in the range $[0, 1]$). Each version provides also the location where to find coverage and test reports in the dataset folders, along with all sources and test directories for each sub-module in the version.

ReCover dataset enriches and consolidates the two popular source RTP datasets from which it has been collected. It adds the full source code, test execution reports and fine-grained per-test coverage reports, in order to reduce the burden of manually collecting and executing projects to retrieve data needed in RTP research. Furthermore, it provides a build environment which can be readily extended to produce different kind of informa-

tion needed to evaluate and execute various prioritization techniques.

5.2 RTP Techniques on ReCover

This section presents an evaluation of several different RTP techniques on pairs of versions present in the *ReCover* dataset, to establish their fault-detection performances on a set of projects with real-world faults. The section initially presents the methodology on which test failures have been mapped to faults, due the absence of fault-localization information, and the experimental setting, defining the techniques employed in the empirical evaluation and the target metrics. Eventually, it describes an analysis of the different outcomes for some sample projects.

5.2.1 Experimental Setting for ReCover

The first point in designing the experimental setting for the evaluation has been to choose the methodology used to build the *failure-to-fault* mapping, i.e., how failures in test cases map to different faults present in the software version. In fact, the information of which faults cause various test cases to fail is not known in the dataset, thus it not possible to tell whether two or more test cases fail for the same defect or not. RTP studies on datasets with real-faults in literature typically choose between two distinct approaches: *all-to-one* and *one-to-one* [113, 122]. The former consists in mapping each failing test cases to a single fault in the software, while the latter treats each failure to be caused by a distinct fault.

Both mappings have their flaws: the all-to-one mapping considers only a single fault in the entire software, which can be plausible when there is a small number of failing test cases; however, as the number of test failures grows, it is highly unlikely that all these failures are caused by a single software defect. This approach typically overestimates the fault-detection capabilities of RTP techniques, as it highly awards permutations scheduling any failing test case in initial positions, disregarding the remainder of the permutations. The one-to-one mapping, on the other hand, presupposes that each test case could not discover more than one fault, and tends to underestimate the rate of fault-detection. In fact, if two test cases exercise the exactly same code, it is highly likely that the failure of one of

these will result in the failure of the other. However, this mapping has the effect that fault-detection rate evaluation metrics assign a higher score to a permutation if such test cases are scheduled one after another, although their subsequent execution provides redundant information to the developers in charge of correcting the defect.

Other intermediate mappings could not be straightforwardly designed, as they involve a fault localization analysis and their subsequent mapping to test cases which exhibit such faults. Due the varying number of test case failures across pairs in the dataset, it has been used as the underlying failure-to-fault mapping for experiments on *ReCover*.

Several RTP techniques have been employed to perform an empirical evaluation on *ReCover*, including coverage-based techniques and different TK-based techniques. One of my study [7] investigated the correlation between different fault-proneness metrics based on various AI models and fault-proneness scores assigned by human experts. A subset of method evolutionary scenarios present in the *ReCover* dataset has been manually labeled by a practitioner and a researcher in software testing and compared with the scores automatically evaluated by the considered AI-based fault-proneness metrics. The outcome of the study suggested that some types of TK functions could be better indicators of fault-proneness of changed methods with respect to the others AI-based metrics (e.g., transformer-based models [132] and *CodeBERT* [45]). This result motivated the inclusion of different variations of the MTK technique, in particular considering either the STK function and the Tree-Kernel distance metric presented in Definition 4.7 to produce the dissimilarity of code fragments. These variations have been evaluated on the dataset along with other defined approaches, illustrated in the following:

- *total*: the traditional total coverage prioritization technique, ordering test cases according to the total number of code elements they cover [114]. The granularity of code elements has been set to statement-level.

- *additional*: a prioritization approach which selects test cases according to the increment on covered code elements they provide [42]. This RTP technique selects the next test which covers the highest number of code elements not yet covered by previous scheduled test cases. As

the total technique, the granularity has been set to statement-level. This permutation typically guarantees high fault-detection rate, but is not commonly employed in practice as it does not scale well with the number of test cases. In fact, once a test case has been selected, the information of code elements not yet covered should be updated for all the remaining tests. For this reason, its time complexity is $O(n^2)$, with n number of test cases in the suite. As the test suite grows, which is typical in RTP practical contexts, the time to evaluate the permutation alone could deplete all testing resources.

- *diff*: the textual difference-based prioritization, also employed in the empirical study described in Chapter 4. It assigns scores to test cases according to the number of changed methods they cover, and re-arranges the permutation in descending order according to these scores.

- *TK dissimilarity*: the MTK technique using the dissimilarity function in Definition 4.6. Two techniques have been employed, according to the inner TK function they employ to evaluate dissimilarity, i.e., STK and PTK.

- *TK distance*: the MTK technique using the Tree-Kernel distance function (presented in Definition 4.7) to evaluate the extent of changes. Also in this case, either STK and PTK have been employed as basis of the distance metric.

- *QS*: the QS approach which re-arranges test cases according to equivalence classes on scores assigned by other prioritization techniques. Three underlying approaches used to score techniques have been employed, the diff approach, TK dissimilarity and TK distance. The latter two internally leverage the PTK function.

The untreated test suite (*NOP*) has been also included to compare the effectiveness of the original execution order of test cases with the permutations produced by the considered RTP techniques.

Fault-detection performances have been evaluated using the APFD metric in Definition 2.2 and the PFF PLF metrics presented in Definition 2.4 and Definition 2.5, respectively.

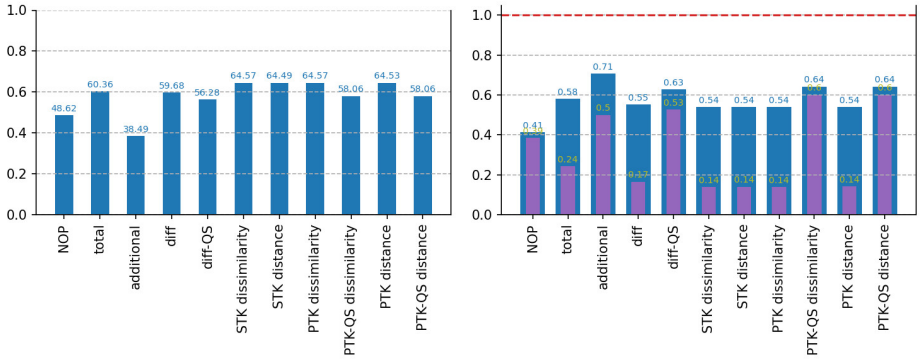


Figure 5.5. Experimental results on an example pair in the *rapidoid@rapidoid* project. APFD values are depicted in the left plot, while PFF (violet) and PLF (blue) results are shown in the right plot.

5.2.2 Results and Discussion

Results obtained through the evaluation of the considered techniques showed fault-detection scores which highly fluctuate across the different pairs in the dataset, for all the techniques. No general trend or pattern resulted from the analysis of these values, and the various techniques performed in significantly different manner both across the various project and for different pairs within the same project. One of the chief factors for these fluctuating results could be ascribed to the lack of information on the location of faults within versions. The approximation through the one-to-one mapping reduces the accuracy of produced fault-detection rates, in particular when there are several failing test cases.

Among all performed experiments, three examples of results are presented in the following. These examples shows different types of fluctuations in the results according to characteristics of subject pairs of versions.

Figure 5.5 shows the APFD, PFF and PLF results for the build with id *310896226* in the project *rapidoid@rapidoid*. There are 5 failing tests in the current version of the experimental pair, which the one-to-one mapping links to as many defects. The *rapidoid@rapidoid* projects is a medium sized project, with a test suite composed by 500 tests. These test cases do not exercise the source thoroughly, as its coverage percentage is around 50%.

APFD results for this scenario shows a low degree of fault-detection rate for all techniques. In particular, it is possible to note that the additional prioritization has worse performance than the total strategy. However, permutation produced by the additional technique tends to have a higher rate of coverage of statements, while the total approach might put test cases stressing the same statements closer in its scheduling. Due to these behaviours, APFD results for these techniques suggests that the one-to-one mapping is not realistic in this case. In fact, test cases only covering already exercised units are scheduled later in the permutation produced by the additional technique, as these tests would not cover any new statement in the software and not showing any new fault even if these tests fail. In the one-to-one mapping, however, this phenomenon strongly penalizes the additional, while enhancing the performance of the total technique, which on the contrary schedules these failing test cases earlier even if their coverage is redundant.

It is possible to see the same behaviour also with all the QS techniques, compared with their counterparts not using the quotient set. All performances are in fact lower, due to the fact that diversify the coverage can delay the execution of failing test cases which possibly uncover already exhibited faults. As a note, diff-QS presents a minor loss in APFD (around 3.5%) with respect to the all TK-based techniques with QS (in which the loss is around 6.5%). In this case, as scores produced by diff are less indicative of same set of covered methods, the equivalence classes are more likely to contain test cases covering different covered elements. On the other hand, scores produced by the TK have a higher chance to represent the same set of covered methods, and thus there is a greater loss in performance. Furthermore, TK-based techniques produce the highest values, with small fluctuations between both the type of TK function used and the dissimilarity/distance employed. The same observations can be deduced by the analysis of the PFF and PLF results.

The other example in Figure 5.6 refers to build *385844174* of the *lukas-krekan@JsonUnit* project. With little more than 4200 lines of code, the project is small. However, it is exhaustively tested, including around 1600 test cases in the current version and a coverage rate greater than 90%. This particular pair of versions presents 50 failures in test cases. For such a small project, this number of failures is hardly due to 50 different defects

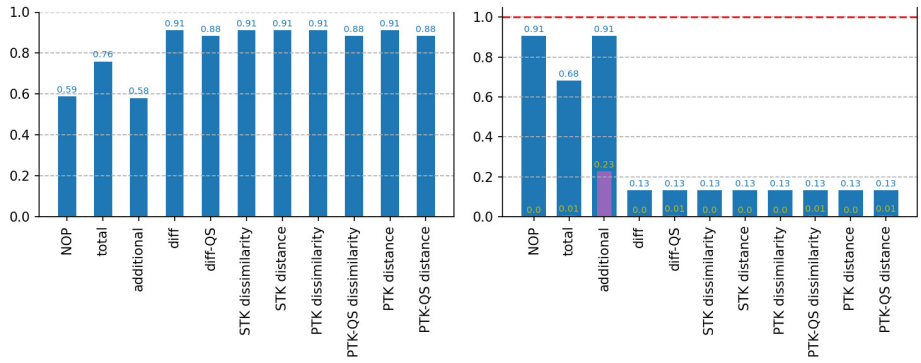


Figure 5.6. Experimental results on an example pair in the *lukas-krekan@JsonUnit* project. APFD values are depicted in the left plot, while PFF (violet) and PLF (blue) results are shown in the right plot.

and the one-to-one mapping cannot realistically represent this scenario. This is suggested also by the amount of changed methods between the two versions, which is around 10.

As it is possible to see from the figure, all difference-based techniques perform similarly, as the produced permutations are very close to each other due to the limited number of changes. Also the loss in APFD values between QS techniques and their counterparts are constant and limited. As it happened in the previous example, also in this scenario the additional technique perform worse than the total, suggesting again that the one-to-one mapping could not be a good approximation of the number of faults in the experiment.

Finally, Figure 5.7 shows APFD, PFF and PLF scores for the build with id *12132071* of the *AdamFisk@LittleProxy* project. The project is one of the smallest in *ReCover* and has less than 100 test cases. In the considered pair of versions, it is present only a single test failure in the current versions. As it is possible to see from the figure, the additional technique is significantly better than the total strategy in all metrics. In this pair, in fact, the problem with the failure-to-fault mapping are not present, as the single failing test can be ascribed to a single fault. The raw churn-based techniques are equivalent to each other and present higher results on all metrics. When applying the QS, however, performances

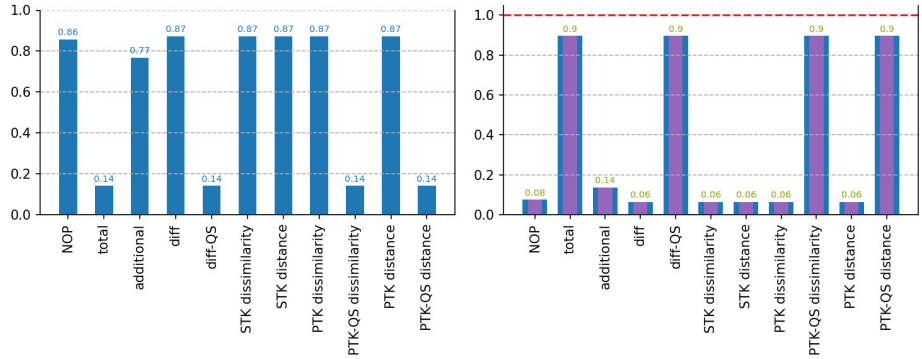


Figure 5.7. Experimental results on an example pair in the *AdamFisk@LittleProxy* project. APFD values are depicted in the left plot, while PFF (violet) and PLF (blue) results are shown in the right plot.

drastically worsen. This is due to the small number of test cases in the current version, which significantly reduces the metrics for small delays in the scheduling. In these approaches, indeed, the failing test case is postponed by the re-arrangement in equivalence classes. Furthermore, in this scenario it is possible to see that the test suite in its original order, denoted with *NOP* in the figure, achieve high results by chance, as the only failing test case is one of the first which is executed by the test runner.

The discussion on the proposed example can be applied to several other scenarios in the *ReCover* dataset. Even if in many pairs of versions the churn-based techniques tend to exhibit high fault detection values, in other cases the performances are lesser than the test suite in its original order. The highest confidence in results is related to scenarios with only one failing test, which nullifies the approximation of the failures-to-faults mapping. However, in these scenarios, all churn-based techniques perform very similarly and the results might not be significant in telling which RTP approach could perform better. On the other hand, more representative scenarios with a higher number of faults suffer from the one-to-one mapping, increasing the inaccuracy on fault-detection performance due their tendency to reward redundancy in permutations. In other terms, real-world faults may produce more feigned results than synthetic faults, if no further information on these faults is supplied.

Chapter 6

Conclusions and Future Works

Regression Test Prioritization is a widely spread practice to reduce the cost of Regression Testing in software evolutionary scenarios, aiming to re-arrange the order of execution of test cases in a permutation which possibly maximizes the rate of fault-detection. RTP is an active topic in software engineering, and several techniques are proposed to prioritize test cases, and their fault-detection performances are assessed through empirical evaluations. Although changes in the source code are one of the principal causes of defects in the software, few studies in the RTP literature consider the code churn to construct a permutation, and these studies typically involve naïve measures of the extent of changes. Furthermore, the lack of commonly accepted datasets and procedures increases the efforts of experimenting in the RTP field.

To address these limitations, the work presented in this thesis focused on two main topics: understanding whether code churn information, along with more refined tools for its evaluation, could benefit RTP activities; and the collection of a dataset which can reduce researchers efforts to experiment and compare different prioritization techniques in uncontrolled environments, encouraging studies in the field.

To this end, I developed Genetic-Diff, a Genetic Algorithm-based RTP technique which searches for a permutation maximizing the rate of coverage of churned parts of source code, as software changes are typically ignored by meta-heuristic approaches to RTP. The fault-detection performance of Genetic-Diff has been empirically evaluated in controlled conditions, i.e.,

on a dataset of well-known projects artificially seeded with synthetic faults. The experimentation showed that Genetic-Diff achieved higher rates of fault-detection with respect to several prioritization techniques commonly employed as baselines in RTP studies. The obtained results suggest that embedding code churn information in meta-heuristic frameworks can be helpful to drive the searching process towards a permutation which can more rapidly detect faults.

My research has been also focused in embedding more refined methods to evaluate code-churn to devise novel RTP techniques. I thus designed MTK, which is driven by the extent of changes by means of Tree Kernel functions on the Abstract Syntax Tree representation of source code, and MTK-QS, an extension of MTK which further re-arranges the test suite with the goal of increasing the rate of code coverage. Both techniques have been empirically evaluated in controlled conditions, in order to analyze both their fault-detection performances and amount of testing resources which could be saved. While MTK do not perform any better than the considered baselines due to redundancy issues in its produced test case scheduling, MTK-QS outperformed all other techniques in terms of fault-detection metrics and showed a significant saving of testing resources. Only in the experimentation of one project, *La4J*, MTK-QS does not produced any benefits, depleting almost all testing resources only to produce the permutation. This issue is due to the nature of the project itself, on which the evaluation of the extent of changes through Tree Kernels has been computationally expensive. This kind of projects is however atypical, and in general MTK-QS produces permutations which can sensibly benefit Regression Testing activities.

The empirical evaluations of Genetic-Diff and MTK-QS suggests the importance of the code churn in the design of prioritization techniques. Nonetheless, considering the extent of code churn alone could not suffice to improve fault-detection performance, as results of MTK seems to indicate. Indeed, including this information in a richer framework, as in Genetic-Diff and MTK-QS, can help to derive a more meaningful permutation of the test cases, increasing the effectiveness of RTP techniques toward higher rates of fault-detection.

Moreover, this thesis presents *ReCover*, a dataset of 114 software evolutionary scenarios among 28 open-source Java projects including real-world

faults. *ReCover* enriches two popular RTP datasets with various auxiliary data, to be readily applicable to a more heterogeneous set of prioritization approaches. The dataset helps in reducing the effort to manually identify and collect software evolutionary scenarios to employ in RTP experimentation. An empirical evaluation has been performed on the *ReCover* dataset, employing several RTP techniques, including variants of the MTK approach. This evaluation showed some limitations in the applicability of such techniques in uncontrolled conditions due the absence of fault-localization information.

Future Works

Different lines of research can arise from topics discussed in this thesis, both *vertically*, involving the further exploitation of different measures of code churn evaluation, and *horizontally*, widening the design of churn-based techniques in different application contexts, each with its specific challenges.

Further Refining Techniques and Tools

In first place, Genetic-Diff considers changes only in a Boolean fashion in its search for a permutation with high coverage rate of churned code. On the other hand, MTK leverages TK functions to produce a refined assessment of the code churn, highlighting source code changes which are more likely to exhibit faults. Although its performance were inadequate in the performed empirical evaluation, the diversification of coverage induced by the application of QS significantly increased the fault-detection rate of its permutations. A complementary GA-based approaches has been outlined, combining the finer evaluation of changes produced by TK within the search for high rate of coverage of churned code embedded in the Genetic-Diff framework. This new technique aims to search a permutation with the highest possible rate of coverage of structural changes, where each change is weighted by the dissimilarity score found by TK. According to preliminary experiments on a limited empirical setting, this combined technique has produced promising results, and an experimentation on a wider set of software evolutionary scenarios should be carried out in order

to better generalize these results.

Concerning the TK-based techniques, MTK-QS produced high fault-detection rates in its empirical evaluation. All the employed tree kernel functions, i.e., STK, SSTK and PTK, search for common fragments in the tree structures on which they are applied, ignoring the contribution of fragments in which any pair of nodes is marked with different labels. However, other TK functions available in literature allow the definition of function which weights the extent of changes in labels. This function can be tuned to produce positive contribution also if labels have changed, in order to further highlight specific changes in the semantic of the respective source code (for example, a `while` loop statement changed to a `for` loop should be more semantically similar than its substitution with a `if` branch). Another line to pursue could be the design of such a weighting function, either manually or through machine learning frameworks, to further refine the evaluation of extent of changes.

The results of different RTP techniques on *ReCover* gave rise to discussions regarding the fact that knowing the precise location of faults could provide a more accurate measure of fault-detection performance in empirical evaluations. To this end, one of the envisioned enrichment of the dataset involves the application of fault-localization techniques in collected projects, to provide further details on the evolutionary scenarios within and mitigate possible threats to validity of experiments on *ReCover*.

Exploring Different Contexts

Nowadays, different software architectural patterns have spread and been employed according to particular business needs. One of the most popular among these patterns is the *Microservice* architecture, a distributed model in which the various components of a software are built and deployed as different stand-alone applications, usually communicating each other through the *HTTP* protocol. These environments set several challenges in testing activities, due to the distributed nature of applications. In particular, the execution of the *end-to-end* tests supposes the instantiation of all service, in order to exercise the interaction with the full system, and the costs to set up the entire infrastructure can be high. To this end, RTP activities can suitably be applied in these contexts to reduce the efforts of the verification phase. I focused on RTP approaches in Microservice-based

contexts during my period abroad at *Technische Universität Wien*, where I designed a churn-based prioritization technique tailored for microservice-based applications, specifically prioritizing end-to-end tests according to changes made to the application endpoints involved in their interactions. The preliminary results of its evaluation on a set of toy projects commonly employed in micro-service research have been promising, justifying a more exhaustive study on real-world project to assess the performance.

Another profitable field of applications of churn-based RTP techniques lays in *Safety-Critical Systems*. As a failure in these systems can produce catastrophic outcomes, they are usually tested thoroughly through formal verification methodologies. These methodologies derive a model to automatically generate the test cases, and then execute them to verify the system. However, the process of generation and execution of test cases is highly demanding, and in scenarios of system evolution could significantly delay the deployment phase of the system. To this end, the extent of changes can be employed to prioritize both processes, in order to confine these costly operations only to more fault-prone changes. A sketch of this idea has been published in one of my papers [9].

Bibliography

[1] Abhilasha and Ashish Sharma. "Test effort estimation in regression testing". In: *Proceedings of the 2013 IEEE International Conference in MOOC, Innovation and Technology in Education, MITE 2013*. IEEE, 2013, pp. 343–348. ISBN: 9781479916269. DOI: 10.1109/MITE.2013.6756364.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd. Pearson Education, Inc, 2006. ISBN: 0-201-10088-6.

[3] Samar Alsaqqa, Samer Sawalha, and Heba Abdel-Nabi. "Agile Software Development: Methodologies and Trends". In: *International Journal of Interactive Mobile Technologies (iJIM)* 14.11 (July 2020), p. 246. ISSN: 1865-7923. DOI: 10.3991/ijim.v14i11.13269.

[4] Sara Alspaugh, Kristen R. Walcott, Michael Belanich, Gregory M. Kapfhammer, and Mary Lou Soffa. "Efficient time-aware prioritization with knapsack solvers". In: *Proceedings of the 1st ACM Int. Workshop on Empirical Assessment of Software Engineering Languages and Technologies, WEASELTech 2007, Held with the 22nd IEEE/ACM Int. Conf. Automated Software Eng., ASE 2007*. Atlanta, Georgia: Association for Computing Machinery (ACM), 2007, pp. 13–18. ISBN: 9781595938800. DOI: 10.1145/1353673.1353676.

[5] Francesco Altiero, Giovanni Colella, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi L.L. Starace. "Change-Aware Regression Test Prioritization using Genetic Algorithms". In: *Proceedings - 2022 48th Euromicro Conference on Software Engineering*

and Advanced Applications (SEAA). IEEE Computer Society, 2022, pp. 125–132. ISBN: 978-1-6654-6152-8. DOI: 10.1109/SEAA56994.2022.00028.

[6] Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi L.L. Starace. "ReCover: A Curated Dataset for Regression Testing Research". In: *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022*. IEEE, 2022, pp. 196–200. DOI: 10.1145/3524842.3528490.

[7] Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "AI-based Fault-proneness Metrics for Source Code Changes". In: *Joint Proceedings of the 32nd International Workshop on Software Measurement (IWSM) and the 17th International Conference on Software Process and Product Measurement (MENSURA)*. Rome, Italy: CEUR, 2023. URL: https://ceur-ws.org/Vol-3543/paper5.pdf.

[8] Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Regression Test Prioritization Leveraging Source Code Similarity with Tree Kernels". In: *Journal of Software: Evolution and Process* (2024), e2653. ISSN: 2047-7473. DOI: 10.1002/smr.2653.

[9] Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Tree Kernels to Support Formal Methods-based Testing of Evolving Specifications". In: *Proceedings of the 4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY) hosted by the 21st International Conference of the Italian Association for Artificial Intelligence (AIxIA 2023)*. Rome, Italy: CEUR, 2023. URL: https://overlay.uniud.it/workshop/2023/papers/paper13.pdf.

[10] Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. *Replication Package for "Regression Test Prioritization Leveraging Source Code Similarity with Tree Kernels"*. Version 1.0. 2023. DOI: 10.5281/zenodo.8256619.

[11] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. "Reinforcement Learning for Test Case Prioritization". In: *IEEE Transactions on Software Engineering* 48.8 (Aug. 2022), pp. 2836–2856. ISSN: 19393520. DOI: 10.1109/TSE.2021.3070549. eprint: 2012.11364.

[12] Anu Bajaj and Om Prakash Sangwan. "A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms". In: *IEEE Access* 7 (2019), pp. 126355–126375. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2938260.

[13] W. Banzhaf. "The "molecular" traveling salesman". In: *Biological Cybernetics* 64.1 (Nov. 1990), pp. 7–14. ISSN: 03401200. DOI: 10.1007/BF00203625.

[14] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. "Code Comparison System based on Abstract Syntax Tree". In: *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. IEEE, Oct. 2010, pp. 668–673. ISBN: 978-1-4244-6769-3. DOI: 10.1109/ICBNMT.2010.5705174.

[15] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. "The cow-suite approach to planning and deriving test suites in UML projects". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2460 LNCS. Dresden, Germany: Springer Verlag, 2002, pp. 383–397. ISBN: 3540442545. DOI: 10.1007/3-540-45800-X_30.

[16] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Signature Series. Pearson Education, 2022. ISBN: 9780137585236.

[17] Moritz Beller, Georgios Gousios, and Andy Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2017, pp. 447–450. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.24.

[18] M. Benerecetti, R. De Guglielmo, U. Gentile, S. Marrone, N. Mazzocca, R. Nardone, A. Peron, L. Velardi, and V. Vittorini. "Dynamic state machines for modelling railway control systems". In: *Science of Computer Programming* 133 (Jan. 2017), pp. 116–153. ISSN: 0167-6423. DOI: 10.1016/J.SCICO.2016.09.002.

[19] Christian Berg, Jens Peter Reus Christensen, and Paul Ressel. *Harmonic Analysis on Semigroups*. Vol. 100. Graduate Texts in Mathematics. New York, NY: Springer New York, 1984. ISBN: 978-1-4612-7017-1. DOI: 10.1007/978-1-4612-1128-0.

[20] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration". In: *Proceedings - International Conference on Software Engineering (ICSE)*. Seoul, South Korea: IEEE Computer Society, June 2020, pp. 1261–1272. ISBN: 9781450371216. DOI: 10.1145/3377811.3380369.

[21] Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, László Langó, and Tibor Gyimóthy. "Code coverage-based regression test selection and prioritization in WebKit". In: *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE. 2012, pp. 46–55. DOI: 10.1109/ICSM.2012.6405252.

[22] David Binkley. "Semantics guided regression test cost reduction". In: *IEEE Transactions on Software Engineering* 23.8 (1997), pp. 498–516. ISSN: 00985589. DOI: 10.1109/32.624306.

[23] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. "Single and Multi-objective Test Cases Prioritization for Self-driving Cars in Virtual Environments". In: *ACM Transactions on Software Engineering and Methodology* 32.2 (Apr. 2023). ISSN: 15577392. DOI: 10.1145/3533818. eprint: 2107.09614.

[24] Tobias Blickle and Lothar Thiele. "A comparison of selection schemes used in evolutionary algorithms". In: *Evolutionary Computation* 4.4 (1996), pp. 361–394. ISSN: 10636560. DOI: 10.1162/EVCO.1996.4.4.361.

[25] C.E. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilità*. Pubblicazioni del R. Istituto superiore di scienze economiche e commerciali di Firenze. Seeber, 1936.

[26] Razvan C Bunescu and Raymond J Mooney. "A shortest path dependency kernel for relation extraction". In: *HLT/EMNLP 2005 - Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*. Association for Computational Linguistics, 2005, pp. 724–731. DOI: 10.3115/1220575.1220666.

[27] Benjamin Busjaeger and Tao Xie. "Learning for test prioritization: an industrial case study". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016, pp. 975–980.

[28] Francesco Camastra and Alessandro Vinciarelli. "Kernel Methods". In: *Machine Learning for Audio, Image and Video Analysis*. Springer, London, 2008. Chap. 9, pp. 211–263. DOI: 10.1007/978-1-84800-007-0_9.

[29] C. Campbell. "An Introduction to Kernel Methods". In: *Radial Basis Function Networks*. Ed. by Robert J. Howlett and Lakhmi C. Jain. Physica, 2001. Chap. 2, pp. 155–192. ISBN: 978-3-7908-2483-4. DOI: 10.1007/978-3-7908-1826-0.

[30] Cagatay Catal and Deepti Mishra. "Test case prioritization: A systematic mapping study". In: *Software Quality Journal* 21.3 (Sept. 2013), pp. 445–478. ISSN: 09639314. DOI: 0.1007/s11219-012-9181-z.

[31] Michael Collins and Nigel Duffy. "Convolution Kernels for Natural Language". In: *Advances in Neural Information Processing Systems 14: Proceedings of the 2001 Conference*. Ed. by Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani. Cambridge, MA, USA: MIT Press, Nov. 2001, pp. 625–632. ISBN: 0262042088. DOI: 10.7551/MITPRESS/1120.003.0085.

[32] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. "A tree Kernel based approach for clone detection". In: *2010 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2010. ISBN: 9781424486298. DOI: 10.1109/ICSM.2010.5609715.

[33] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. "What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks". In: *IEEE Transactions on Software Engineering* 47.7 (July 2021), pp. 1452–1467. ISSN: 0098-5589. DOI: 10.1109/TSE.2019.2925345.

[34] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. "Scalable Approaches for Test Suite Reduction". In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, May 2019, pp. 419–429. ISBN: 9781728108698. DOI: 10.1109/ICSE.2019.00055.

[35] Lawrence Davis. "Applying Adaptive Algorithms to Epistatic Domains". In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'85. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1985, pp. 162–164. ISBN: 0934613028.

[36] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system". In: *Software Testing, Verification and Reliability* 25.4 (June 2015), pp. 371–396. ISSN: 0960-0833. DOI: 10.1002/stvr.1572.

[37] Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. "A Test Case Prioritization Genetic Algorithm Guided by the Hypervolume Indicator". In: *IEEE Transactions on Software Engineering* 46.6 (June 2020), pp. 674–696. ISSN: 19393520. DOI: 10.1109/TSE.2018.2868082.

[38] Torgeir Dingsøyr, Sridhar Nerur, Venugopal Balijepally, and Nils Brede Moe. "A decade of agile methodologies: Towards explaining agile software development". In: *Journal of Systems and Software* 85.6 (June 2012), pp. 1213–1221. ISSN: 01641212. DOI: 10.1016/j.jss.2012.02.033.

[39] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact". In: *Empirical Software Engineering* 10.4 (Oct. 2005), pp. 405–435. ISSN: 1382-3256. DOI: 10.1007/s10664-005-3861-2.

[40] Hyunsook Do and Gregg Rothermel. "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques". In: *IEEE Transactions on Software Engineering* 32.9 (Sept. 2006), pp. 733–752. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.92.

[41] Sebastian Elbaum, Praveen Kallakuri, Alexey Malishevsky, Gregg Rothermel, and Satya Kanduri. "Understanding the effects of changes on the cost-effectiveness of regression testing techniques". In: *Software Testing Verification and Reliability* 13.2 (Apr. 2003), pp. 65–83. ISSN: 1099-1689. DOI: 10.1002/STVR.263.

[42] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. "Incorporating varying test costs and fault severities into test case prioritization". In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*. IEEE Computer Society, 2001, pp. 329–338. ISBN: 0769510507. DOI: 10.1109/ICSE.2001.919106.

[43] Emelie Engström, Per Runeson, and Andreas Ljung. "Improving Regression Testing Transparency and Efficiency with History-Based Prioritization – An Industrial Case Study". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, Mar. 2011, pp. 367–376. ISBN: 978-1-61284-174-8. DOI: 10.1109/ICST.2011.27.

[44] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit*. Wiley Publishing, 2003, p. 552. ISBN: 0471463612.

[45] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

[46] Simone Filice, Giuseppe Castellucci, Giovanni Da San Martino, Alessandro Moschitti, Danilo Croce, and Roberto Basili. "KELP: a Kernel-based Learning Platform". In: *Journal of Machine Learning Research* 18.191 (2018), pp. 1–5. URL: http://jmlr.org/papers/v18/16-087.html.

[47] E Gamma, R Helm, R Johnson, and J Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Pearson Education, 1994. ISBN: 9780321700698.

[48] Dongdong Gao, Xiangying Guo, and Lei Zhao. "Test case prioritization for regression testing based on ant colony optimization". In: *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS). 2015, pp. 275–279. DOI: 10.1109/ICSESS.2015.7339054.

[49] Thomas Gärtner. "A survey of kernels for structured data". In: *ACM SIGKDD Explorations Newsletter* 5.1 (July 2003), pp. 49–58. ISSN: 1931-0145. DOI: 10.1145/959242.959248.

[50] David E Goldberg and Robert Lingle. "Alleles, Loci and the Traveling Salesman Problem". In: *Proceedings of the 1st International Conference on Genetic Algorithms*. USA: L. Erlbaum Associates Inc., 1985, pp. 154–159. ISBN: 0805804269.

[51] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Mutations: How Close are they to Real Faults?" In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, Nov. 2014, pp. 189–200. ISBN: 978-1-4799-6033-0. DOI: 10.1109/ISSRE.2014.40.

[52] Georgios Gousios. "The GHTorent dataset and tool suite". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, May 2013, pp. 233–236. ISBN: 978-1-4673-2936-1. DOI: 10.1109/MSR.2013.6624034.

[53] John J. Greffenstette and James E. Baker. "How Genetic Algorithms Work: A Critical Look at Implicit Parallelism". In: *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 20–27. DOI: 10.5555/645512.657235.

[54] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. "A Unified Test Case Prioritization Approach". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.2 (Dec. 2014). ISSN: 15577392. DOI: 10.1145/2685614.

[55] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. "To Be Optimal or Not in Test-Case Prioritization". In: *IEEE Transactions on Software Engineering* 42.5 (May 2015), pp. 490–504. ISSN: 00985589. DOI: 10.1109/TSE.2015.2496939.

[56] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A methodology for controlling the size of a test suite". In: *Proceedings - Conference on Software Maintenance*. IEEE, Nov. 1990, pp. 302–310. ISBN: 0818620919. DOI: 10.1109/ICSM.1990.131378.

[57] Muhammad Hasnain, Muhammad Fermi Pasha, Chern Hong Lim, and Imran Ghan. "Recurrent neural network for web services performance forecasting, ranking and regression testing". In: *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference, APSIPA ASC 2019*. Institute of Electrical and Electronics Engineers Inc., Nov. 2019, pp. 96–105. DOI: 10.1109/APSIPAASC47483.2019.9023052.

[58] David Haussler. *Convolution Kernels on Discrete Structures*. Tech. rep. Santa Cruz, California, US: University of California, 1999.

[59] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. "Comparing white-box and black-box test prioritization". In: *Proceedings - International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2016, pp. 523–534. ISBN: 9781450339001. DOI: 10.1145/2884781.2884791.

[60] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. "Kernel methods in machine learning". In: *Annals of Statistics* 36.3 (June 2008), pp. 1171–1220. DOI: 10.1214/009053607000000677.

[61] John H Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975. ISBN: 9780262275552. DOI: 10.7551/mitpress/1090.001.0001.

[62] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang. "A history-based cost-cognizant test case prioritization technique in regression testing". In: *Journal of Systems and Software* 85.3 (2012), pp. 626–637.

[63] Rubing Huang, Quanjun Zhang, Dave Towey, Weifeng Sun, and Jinfu Chen. "Regression test case prioritization by code combinations coverage". In: *Journal of Systems and Software* 169 (2020), p. 110712. DOI: 10.1016/j.jss.2020.110712.

[64] Yu Chi Huang, Chin Yu Huang, Jun Ru Chang, and Tsan Yuan Chen. "Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history". In: *Proceedings of the 34th IEEE International Computer Software and Applications Conference*. IEEE Computer Society, 2010, pp. 413–418. DOI: 10.1109/COMPSAC.2010.66.

[65] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. "Change-aware preemption prioritization". In: *2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings*. ACM, July 2011, pp. 133–143. ISBN: 9781450305624. DOI: 10.1145/2001420.2001437.

[66] Hosney Jahan, Ziliang Feng, S. M.Hasan Mahmud, and Penglin Dong. "Version specific test case prioritization approach based on artificial neural network". In: *Journal of Intelligent & Fuzzy Systems* 36.6 (June 2019), pp. 6181–6194. ISSN: 18758967. DOI: 10.3233/JIFS-181998.

[67] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. "Adaptive random test case prioritization". In: *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Auckland, New Zealand: IEEE, 2009, pp. 233–244. ISBN: 9780769538914. DOI: 10.1109/ASE.2009.77.

[68] James A Jones and Mary Jean Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage". In: *IEEE Transactions on software Engineering* 29.3 (2003), pp. 195–209.

[69] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: a database of existing faults to enable controlled testing studies for Java programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, July 2014, pp. 437–440. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055.

[70] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. "Are mutants a valid substitute for real faults in software testing?" In: *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Vol. 16-21-Nove. Association for Computing Machinery, Nov. 2014, pp. 654–665. ISBN: 9781450330565. DOI: 10.1145/2635868.2635929.

[71] Arvinder Kaur and Divya Bhatt. "Hybrid Particle Swarm Optimization for Regression Testing". In: *International Journal on Computer Science and Engineering* 3 (May 2011).

[72] Jung-Min Kim and Adam Porter. "A history-based test prioritization technique for regression testing in resource constrained environments". In: *Proceedings of the 24th international conference on software engineering*. Proceedings of the 24th international conference on software engineering (ICSE). 2002, pp. 119–129.

[73] Bogdan Korel and George Koutsogiannakis. "Experimental comparison of code-based and model-based test prioritization". In: *2009 International Conference on Software Testing, Verification, and Validation Workshops*. 2009 International Conference on Software Testing, Verification, and Validation Workshops. 2009, pp. 77–84.

[74] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. "Model-based test prioritization heuristic methods and their evaluation". In: *Proceedings of the 3rd International Workshop Advances in Model Based Testing, AMOST 2007*. 2007, pp. 34–43. ISBN: 9781595938503. DOI: 10.1145/1291535.1291539.

[75] Bogdan Korel, Luay H. Tahat, and Mark Harman. "Test prioritization using system models". In: *Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM*. IEEE, 2005, pp. 559–568. DOI: 10.1109/ICSM.2005.87.

[76] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. "Measuring the Cost of Regression Testing in Practice". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. 2017, pp. 821–830. ISBN: 9781450351058. DOI: 10.1145/3106237.3106288.

[77] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. "System-Level Test Case Prioritization Using Machine Learning". In: *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, Dec. 2016, pp. 361–368. DOI: 10.1109/ICMLA.2016.0065.

[78] P. Larrañaga, C. M.H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. "Genetic algorithms for the travelling salesman problem: A review of representations and operators". In: *Artificial Intelligence Review* 13.2 (1999), pp. 129–170. ISSN: 02692821. DOI: 10.1023/A:1006529012972.

[79] Zheng Li, Mark Harman, and Robert M. Hierons. "Search Algorithms for Regression Test Case Prioritization". In: *IEEE Transactions on Software Engineering* 33.4 (2007), pp. 225–237. DOI: 10.1109/TSE.2007.38.

[80] Jackson A.Prado Lima and Silvia Regina Vergilio. "A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments". In: *IEEE Transactions on Software Engineering* 48.2 (Feb. 2022), pp. 453–465. ISSN: 19393520. DOI: 10.1109/TSE.2020.2992428.

[81] Chu-Ti Lin, Cheng-Ding Chen, Chang-Shi Tsai, and Gregory M Kapfhammer. "History-based test case prioritization with software version awareness". In: *2013 18th International Conference on Engineering of Complex Computer Systems*. 2013 18th International Conference on Engineering of Complex Computer Systems. 2013, pp. 171–172.

[82] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. "How does regression test prioritization perform in real-world software evolution?" In: *Proceedings - International Conference on Software Engineering (ICSE)*. Vol. 14-

22-May. ACM, May 2016, pp. 535–546. ISBN: 9781450339001. DOI: 10.1145/2884781.2884874.

[83] Q. Luo, K. Moran, D. Poshyvanyk, and L. Zhang. "How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects". In: *IEEE Transactions on Software Engineering* 45.11 (2019). ISSN: 00985589. DOI: 10.1109/TSE.2018.2822270.

[84] H. B. Mann and D. R. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other". In: *Ann. Math. Statist.* 18.1 (Mar. 1947), pp. 50–60. ISSN: 0003-4851. DOI: 10.1214/AOMS/1177730491.

[85] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. "Introduction to Information Retrieval". In: Cambridge, UK: Cambridge University Press, 2008, pp. 61–62. ISBN: 978-0-521-86571-5.

[86] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *2013 IEEE International Conference on Software Maintenance*. IEEE, Sept. 2013, pp. 540–543. ISBN: 978-0-7695-4981-1. DOI: 10.1109/ICSM.2013.91.

[87] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.

[88] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. "RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, June 2020, pp. 385–396. ISBN: 9781450375177. DOI: 10.1145/3379597.3387458.

[89] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. "A static approach to prioritizing JUnit test cases". In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1258–1275. ISSN: 00985589. DOI: 10.1109/TSE.2011.106.

[90] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. "Taming Google-scale continuous testing". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, May 2017, pp. 233–242. ISBN: 978-1-5386-2717-4. DOI: 10.1109/ICSE-SEIP.2017.16.

[91] Siavash Mirarab, Soroush Akhlaghi, and Ladan Tahvildari. "Size-constrained regression test case selection using multicriteria optimization". In: *IEEE Transactions on Software Engineering* 38.4 (2012), pp. 936–956. ISSN: 00985589. DOI: 10.1109/TSE.2011.56.

[92] Nikolaos Mittas and Lefteris Angelis. "Ranking and clustering software cost estimation models through a multiple comparisons algorithm". In: *IEEE Transactions on software engineering* 39.4 (2012), pp. 537–551.

[93] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. "Exploring test suite diversification and code coverage in multi-objective test case selection". In: *Proceedings - IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015*. Institute of Electrical and Electronics Engineers Inc., May 2015. ISBN: 9781479971251. DOI: 10.1109/ICST.2015.7102588.

[94] Alessandro Moschitti. "Efficient convolution kernels for dependency and constituent syntactic trees". In: *Machine Learning: ECML 2006. Lecture Notes in Computer Science()*. Ed. by J. Fürnkranz, T. Scheffer, and M. Spiliopoulou. Vol. 4212 LNAI. Springer, Berlin, Heidelberg, 2006, pp. 318–329. ISBN: 354045375X. DOI: 10.1007/11871842_32.

[95] Alessandro Moschitti. "Making Tree Kernels Practical for Natural Language Learning". In: *11th Conference of the European Chapter of the Association for Computational Linguistics*. Ed. by Shuly Wintner Diana McCarthy. Trento, Italy: Association for Computational Linguistics, Apr. 2006, pp. 113–120. URL: https://aclanthology.org/E06-1015.

[96] Rajendrani Mukherjee and K. Sridhar Patnaik. "A survey on different approaches for software test case prioritization". In: *Journal of King Saud University - Computer and Information Sciences* 33.9

(Nov. 2021), pp. 1041–1054. ISSN: 1319-1578. DOI: 10.1016/J.JKSUCI.2018.09.005.

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

[98] Soumen Nayak, Chiranjeev Kumar, Sachin Tripathi, Nirjharini Mohanty, and Vishal Baral. "Regression test optimization and prioritization using Honey Bee optimization algorithm with fuzzy rule base". In: *Soft Computing* (2020), pp. 1–18.

[99] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. "Understanding source code evolution using abstract syntax tree matching". In: *Proceedings of the 2005 international workshop on Mining software repositories - MSR '05*. New York, New York, USA: ACM Press, 2005, pp. 1–5. ISBN: 1595931236. DOI: 10.1145/1083142.1083143.

[100] Truc Vien T. Nguyen, Alessandro Moschitti, and Giuseppe Riccardi. "Convolution kernels on constituent, dependency and sequential structures for relation extraction". In: *EMNLP 2009 - Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: A Meeting of SIGDAT, a Special Interest Group of ACL, Held in Conjunction with ACL-IJCNLP 2009*. Association for Computational Linguistics, 2009, pp. 1378–1387. DOI: 10.3115/1699648.1699684.

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

[102] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. "Test case selection and prioritization using machine learning: a systematic literature review". In: *Empirical Software Engineering* 27.2 (Mar. 2022), pp. 1–43. ISSN: 15737616. DOI: 10.1007/s10664-021-10066-6. eprint: 2106.13891.

[103] Chhabi Rani Panigrahi and Rajib Mall. "Model-based regression test case prioritization". In: *Communications in Computer and Information Science*. Vol. 54. Springer, Berlin, Heidelberg, 2010, pp. 380–

385. ISBN: 9783642120343. DOI: 10.1007/978-3-642-12035-0_39/COVER.

[104] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. "Mutation Testing Advances: An Analysis and Survey". In: *Advances in Computers* 112 (Jan. 2019), pp. 275–378. ISSN: 0065-2458. DOI: 10.1016/BS.ADCOM.2018.03.015.

[105] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik. "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing". In: *Proceedings of the 2nd IEEE International Conference on Secure System Integration and Reliability Improvement, SSIRI 2008*. Yokohama, Japan: IEEE, July 2008, pp. 39–46. ISBN: 9780769532660. DOI: 10.1109/SSIRI.2008.52.

[106] David Paterson, Jose Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. "An Empirical Study on the Use of Defect Prediction for Test Case Prioritization". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Apr. 2019, pp. 346–357. ISBN: 978-1-7281-1736-2. DOI: 10.1109/ICST.2019.00041.

[107] Qianyang Peng, August Shi, and Lingming Zhang. "Empirically revisiting and enhancing IR-based test-case prioritization". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020, pp. 324–336. DOI: 10.1145/3395363.3397383.

[108] Jay M. Ponte and W. Bruce Croft. "A Language Modeling Approach to Information Retrieval". In: *SIGIR 1998 - Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, Inc, Aug. 1998, pp. 275–281. ISBN: 9781581130157. DOI: 10.1145/290941.291008.

[109] Václav Rajlich. "Software evolution and maintenance". In: *FOSE 2014: Future of Software Engineering Proceedings (FOSE)*. New York, NY, USA: ACM, May 2014, pp. 133–144. ISBN: 9781450328654. DOI: 10.1145/2593882.2593893.

[110] Lukas Rosenbauer, Anthony Stein, Roland Maier, David Pätzel, and Jörg Hähner. "XCS as a reinforcement learning approach to automatic test case prioritization". In: *GECCO 2020 Companion - Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, Inc, July 2020, pp. 1798–1806. ISBN: 9781450371278. DOI: 10.1145/3377929.3398128.

[111] Gregg Rothermel and Mary Jean Harrold. "A safe, efficient regression test selection technique". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.2 (Apr. 1997), pp. 173–210. ISSN: 1049331X. DOI: 10.1145/248233.248262.

[112] Gregg Rothermel and Mary Jean Harrold. "Analyzing regression test selection techniques". In: *IEEE Transactions on Software Engineering* 22.8 (1996), pp. 529–551. ISSN: 00985589. DOI: 10.1109/32.536955.

[113] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. "Empirical study of the effects of minimization on the fault detection capabilities of test suites". In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. Bethesda, MD: IEEE, 1998, pp. 34–43. DOI: 10.1109/ICSM.1998.738487.

[114] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. "Test case prioritization: an empirical study". In: *Proceedings of the IEEE Conference on Software Maintenance*. Oxford, UK: IEEE, 1999, pp. 179–188. DOI: 10.1109/ICSM.1999.792604.

[115] Gregg Rothermel, Roland H. Untcn, Chengyun Chu, and Mary Jean Harrold. "Prioritizing test cases for regression testing". In: *IEEE Transactions on Software Engineering* 27.10 (2001), pp. 929–948. ISSN: 00985589. DOI: 10.1109/32.962562.

[116] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. "An information retrieval approach for regression test prioritization based on program changes". In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Vol. 1. IEEE Computer Society, May 2015, pp. 268–279. DOI: 10.1109/ICSE.2015.47.

[117] Gerard Salton and Christopher Buckley. "Term-weighting approaches in automatic text retrieval". In: *Information Processing & Management* 24.5 (Jan. 1988), pp. 513–523. ISSN: 0306-4573. DOI: 10.1016/0306-4573(88)90021-0.

[118] Federica Sarro, Alessio Petrozziello, and Mark Harman. "Multi-Objective Software Effort Estimation". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ICSE '16. ICSE'16. Austin, Texas: Association for Computing Machinery, 2016, pp. 619–630. ISBN: 9781450339001. DOI: 10.1145/2884781.2884830.

[119] Bernhard Schölkopf. "The Kernel trick for distances". In: *NIPS'00: Proceedings of the 13th International Conference on Neural Information Processing Systems*. MIT Press, 2000, pp. 283–289. DOI: 10.5555/3008751.3008793.

[120] Mali Senapathi, Jim Buchan, and Hady Osman. "DevOps Capabilities, Practices, and Challenges". In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. New York, NY, USA: ACM, June 2018, pp. 57–67. ISBN: 9781450364034. DOI: 10.1145/3210459.3210465.

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

[122] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. "Evaluating test-suite reduction in real software evolution". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, July 2018, pp. 84–94. ISBN: 9781450356992. DOI: 10.1145/3213846.3213875.

[123] Kilho Shin and Tetsuji Kuboyama. "A Generalization of Haussler's Convolution Kernel-Mapping Kernel". In: *Proceedings of the 25th International Conference on Machine Learning (ICML)*. Helsinki, Finland: Association for Computing Machinery, 2008, pp. 944–951. DOI: 10.1145/1390156.1390275.

[124] Ajmer Singh, Anita Singhrova, and Rajesh Kumar Bhatia. "Machine Learning based Test Case Prioritization in Object Oriented Testing". In: *International Journal of Recent Technology and Engineering* 8.3 (2019), pp. 2277–3878. DOI: 10.35940/ijrte.C3968.098319.

[125] I Sommerville. *Software Engineering*. 10th. Boston, Massachusetts, USA: Pearson, 2015, p. 816. ISBN: 978-0-13-703515-1.

[126] Daniel Staegemann, Matthias Volk, Erik Lautenschlager, Matthias Pohl, Mohammad Abdallah, and Klaus Turowski. "Applying Test Driven Development in the Big Data Domain – Lessons From the Literature". In: *2021 International Conference on Information Technology (ICIT)*. Vol. 31. 1. IEEE, July 2021, pp. 511–516. ISBN: 978-1-6654-2870-5. DOI: 10.1109/ICIT52682.2021.9491728.

[127] Richard Stutzke. *Estimating software-intensive systems: projects, products, and processes*. Addison-Wesley Professional, 2005, p. 944. ISBN: 978-0-7686-8523-7.

[128] Baswaraju Swathi. "Automated Test Case Prioritization and Evaluation using Genetic Algorithm". In: *2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS)*. IEEE, June 2022, pp. 1–5. DOI: 10.1109/IC3SIS54991.2022.9885535.

[129] "Test Case Selection: A Systematic Literature Review". In: *International Journal of Software Engineering and Knowledge Engineering* 24.4 (Sept. 2014), pp. 653–676. ISSN: 02181940. DOI: 10.1142/S0218194014500259.

[130] Vinita Tomar, Mamta Bansal, and Pooja Singh. "Regression Testing Approaches, Tools, and Applications in Various Environments". In: *2022 4th International Conference on Artificial Intelligence and Speech Technology (AIST)*. Delhi, India: IEEE, Dec. 2022, pp. 1–6. ISBN: 978-1-6654-9902-6. DOI: 10.1109/AIST55798.2022.10064753.

[131] Andreas Van Cranenburgh. "Extraction of phrase-structure fragments with a linear average time tree-kernel". In: *Computational Linguistics in the Netherlands Journal* 4 (Dec. 2014), pp. 3–16. ISSN: 22114009. URL: https://clinjournal.org/clinj/article/view/36.

[132] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in Neural Information Processing Systems*. Vol. 2017-Decem. 2017, pp. 5999–6009. DOI: 10.5555/3295222.3295349.

[133] M. Vidoni. "A systematic process for Mining Software Repositories: Results from a systematic literature review". In: *Information and Software Technology* 144 (Apr. 2022). ISSN: 09505849. DOI: 10.1016/j.infsof.2021.106791.

[134] Vinod Chandra Vinod and Anand H. S. "Nature inspired meta heuristic algorithms for optimization problems". In: *Computing* 104.2 (Feb. 2021), pp. 251–269. ISSN: 14365057. DOI: 10.1007/S00607-021-00955-5.

[135] Manish Virmani. "Understanding DevOps & bridging the gap from continuous integration to continuous delivery". In: *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*. IEEE, May 2015, pp. 78–82. ISBN: 978-1-4673-7551-1. DOI: 10.1109/INTECH.2015.7173368.

[136] S.V.N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. "Graph Kernels". In: *Journal of Machine Learning Research* 11.40 (2010), pp. 1201–1242. URL: https://jmlr.org/papers/v11/vishwanathan10a.html.

[137] Nancy J Wahl. "An Overview of Regression Testing". In: *SIGSOFT Softw. Eng. Notes* 24.1 (1999), pp. 69–73. ISSN: 0163-5948. DOI: 10.1145/308769.308790.

[138] Mengqiu Wang. "A Re-examination of dependency path kernels for relation extraction". In: *IJCNLP 2008 - 3rd International Joint Conference on Natural Language Processing, Proceedings of the Conference*. Vol. 2. Association for Computational Linguistics, 2008, pp. 841–846. URL: https://aclanthology.org/I08-2119.

[139] Darrell Whitley. "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best". In: *Proceedings of the Third International Conference on Genetic Algorithms*. George Mason University, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 116–121. DOI: 10.5555/93126.93169.

[140] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Vol. 9783642290442. Springer Berlin Heidelberg, July 2012, pp. 1–236. ISBN: 9783642290442. DOI: 10.1007/978-3-642-29044-2/COVER.

[141] Ying Xing, Xingde Wang, and Qianpeng Shen. "Test case prioritization based on Artificial Fish School Algorithm". In: *Computer Communications* 180 (2021), pp. 295–302. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2021.09.014.

[142] Dharmveer Kumar Yadav and Sandip Dutta. "Regression Test Case Prioritization Technique Using Genetic Algorithm". In: *ICCI 2015: Advances in Computational Intelligence*. Springer, Singapore, 2017, pp. 133–140. DOI: 10.1007/978-981-10-2525-9_13.

[143] Fang Yuan, Yi Bian, Zheng Li, and Ruilian Zhao. "Epistatic genetic algorithm for test case prioritization". In: *Lecture Notes in Computer Science*. Ed. by Search-Based Software Engineering. SSBSE 2015. Vol. 9275. Springer International Publishing, 2015, pp. 109–124. ISBN: 9783319221823. DOI: 10.1007/978-3-319-22183-0_8/TABLES/6.

[144] Ma Zengkai and Zhao Jianjun. "Test case prioritization based on analysis of program structure". In: *Proceedings of the 15th Asia-Pacific Software Engineering Conference*. IEEE, Dec. 2008, pp. 471–478. DOI: 10.1109/APSEC.2008.63.

[145] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. "Understanding bag-of-words model: a statistical framework". In: *International Journal of Machine Learning and Cybernetics* 1.1-4 (Dec. 2010), pp. 43–52. ISSN: 1868-8071. DOI: 10.1007/s13042-010-0001-0.

Author's publications

Journal Papers

1. Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Regression Test Prioritization Leveraging Source Code Similarity with Tree Kernels". In: *Journal of Software: Evolution and Process* (2024), e2653. ISSN: 2047-7473. DOI: 10.1002/smr.2653.

Conference Papers

1. Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "AI-based Fault-proneness Metrics for Source Code Changes". In: *Joint Proceedings of the 32nd International Workshop on Software Measurement (IWSM) and the 17th International Conference on Software Process and Product Measurement (MENSURA)*. Rome, Italy: CEUR, 2023. URL: https://ceur-ws.org/Vol-3543/paper5.pdf.

2. Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Tree Kernels to Support Formal Methods-based Testing of Evolving Specifications". In: *Proceedings of the 4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY) hosted by the 21st International Conference of the Italian Association for Artificial In-*

telligence (AIxIA 2023). Rome, Italy: CEUR, 2023. URL: https://overlay.uniud.it/workshop/2023/papers/paper13.pdf.

3. Francesco Altiero, Giovanni Colella, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi L.L. Starace. "Change-Aware Regression Test Prioritization using Genetic Algorithms". In: *Proceedings - 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE Computer Society, 2022, pp. 125–132. ISBN: 978-1-6654-6152-8. DOI: 10.1109/SEAA56994.2022.00028.

4. Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi L.L. Starace. "ReCover: A Curated Dataset for Regression Testing Research". In: *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022*. IEEE, 2022, pp. 196–200. DOI: 10.1145/3524842.3528490.

5. Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Inspecting Code Churns to Prioritize Test Cases". In: *Proceedings of Testing Software and Systems: 32nd IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2020)*. Vol. 12543 LNCS. Naples, Italy: Springer Science and Business Media Deutschland GmbH, 2020, pp. 272–285. ISBN: 9783030648800. DOI: 10.1007/978-3-030-64881-7_17.