Regression test prioritization leveraging source code similarity with tree kernels

Regression test prioritization leveraging source code similarity with tree kernels

Francesco Altiero[1] | **Anna Corazza**[1] | **Sergio Di Martino**[1] | **Adriano Peron**[2] | **Luigi Libero Lucio Starace**[1]

[1]Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione, Università degli Studi di Napoli Federico II, Naples, Italy

[2]Dipartimento di Matematica e GeoscienzeUniversità degli Studi di Trieste, Trieste, Italy

Correspondence
Francesco Altiero and Luigi Libero Lucio Starace, DIETI, Università degli Studi di Napoli Federico II, Naples, Italy.
Email: francesco.altiero@unina.it and luigiliberolucio.starace@unina.it

Abstract

Regression test prioritization (RTP) is an active research field, aiming at re-ordering the tests in a test suite to maximize the rate at which faults are detected. A number of RTP strategies have been proposed, leveraging different factors to reorder tests. Some techniques include an analysis of changed source code, to assign higher priority to tests stressing modified parts of the codebase. Still, most of these change-based solutions focus on simple text-level comparisons among versions. We believe that measuring source code changes in a more refined way, capable of discriminating between mere textual changes (e.g., renaming of a local variable) and more structural changes (e.g., changes in the control flow), could lead to significant benefits in RTP, under the assumption that major structural changes are also more likely to introduce faults. To this end, we propose two novel RTP techniques that leverage *tree kernels* (TK), a class of similarity functions largely used in Natural Language Processing on tree-structured data. In particular, we apply TKs to abstract syntax trees of source code, to more precisely quantify the extent of structural changes in the source code, and prioritize tests accordingly. We assessed the effectiveness of the proposals by conducting an empirical study on five real-world Java projects, also used in a number of RTP-related papers. We automatically generated, for each considered pair of software versions (i.e., old version, new version) in the evolution of the involved projects, 100 variations with artificially injected faults, leading to over 5k different software evolution scenarios overall. We compared the proposed prioritization approaches against well-known prioritization techniques, evaluating both their effectiveness and their execution times. Our findings show that leveraging more refined code change analysis techniques to quantify the extent of changes in source code can lead to relevant improvements in prioritization effectiveness, while typically introducing negligible overheads due to their execution.

KEYWORDS
regression testing, source code changes, test prioritization, tree kernels

1 | INTRODUCTION

With modern software development practices, such as *continuous integration/delivery*, new versions are generated with high frequency, and often, these updated versions should be also readily deployed. Within this evolutionary process, it is crucial to make sure that the code changes introduced by developers do not adversely impact existing functionalities, that is, that no *regression fault* has been introduced. *Regression testing* is the set of activities aimed at providing confidence that no fault makes its way into production code, causing unintended behaviors in the application.[1]

The most straightforward approach to regression testing consists in the *retest all* strategy, that is, in running the entire test suite of the software after each evolution step.[2] This, however, may not be always feasible, as the execution of the entire test suite can be prohibitively time- and resource-consuming. Moreover, recent trends in software engineering moving towards shorter software development life-cycle, such as *continuous delivery*,[3] further complicate the situation by imposing stricter restrictions and constraints on how regression testing can be performed within limited resources.[4,5]

To overcome these issues and ease regression testing efforts, different strategies have been proposed in the literature. In particular, *regression test prioritization* (*RTP*) has been widely investigated and is among the most used practices in industrial contexts. RTP approaches aim to produce a permutation of the test suite which can be meaningful and more advantageous to testing purposes. This practice maximizes the effectiveness of the testing activities even when the expiration of constraints suddenly interrupts the test execution.[6]

The goal of RTP techniques is the discovery of faults as soon as possible, and several strategies evaluate different properties of the software under test as *proxy* measures towards this goal. These exploited properties are manifold and include *test coverage* information, *software quality metrics*, and data on *software history*. However, few studies take into account changes in the codebase between the previous version of the software and the new release under test. Furthermore, the evaluation of changes is typically performed by naive measures, often applying simple textual distance algorithms to the purpose. On the other hand, more refined approaches, such as those applied in domains like *code change impact analysis* and *code-based defect prediction*, are not easily applicable due to the strict temporal constraint which RTP implies. For example, techniques based on concept lattice[7,8] often introduce time overheads which are not applicable to the domain of RTP, due to their tight scaling with the size of projects. Other techniques relying on deep networks[9–11] had proven useful in defecting prediction tasks. However, their models typically need a long training time on a specific project before being effectively used. In some cases, these models also need other information regarding the changes, for example, meaningful commit messages with precise descriptions by the developers, that are not always available in practice. For this reason, in this paper, we propose two novel test case prioritization approaches that exploit measures which quantify the impact of code changes among consecutive releases of a software product. These measures do not directly depend on the size of the software and typically introduce negligible overheads, thus being feasibly applicable to regression testing activities in several cases.

The first approach we propose leverages a combination of traditional test coverage metrics, software evolution information, and a refined analysis of the changes in the structure of the source code. Intuitively, in presence of code modifications, we aim at prioritizing test cases exercising the changes that are more fault-prone (e.g., structural changes in the control flow), while delaying the execution of those that are less likely to introduce errors (such as refactoring operation involving the renaming of a local variable). To do so, we exploit the *abstract syntax tree* (AST) representation of source code and quantify the degree of structural changes between subsequent software versions by means of tree kernel (TK) functions. TKs are a type of similarity measure for tree-based data structures, which were proved to be very effective in software engineering domains.[12,13] We designed this approach, namely, the *method-level tree kernel prioritization* (*MTK*), to assign a *score* to a test case in the suite according to the sum of the degree of changes of the methods it covers. Subsequently, the test cases in the suite are sorted in descending order of such assigned scores. The approach, however, tends to exhibit *redundancy*, scheduling several test cases which uncover the same set of faults one after another, and then reducing the rate of fault detection of the permutation. In fact, in the reordering of test cases according to their scores, the resulting schedule of MTK can possibly lead to *redundancy* in coverage of changed methods, slowing down the discovery of faults located in different, yet uncovered, parts of the code.

To overcome the issue, we designed a further technique which is an evolution of the MTK approach, that is, *method-level tree kernel with quotient set* (*MTK-QS*). This approach is based on the intuition that test cases covering the same set of methods sum up to the same score assigned by MTK and have high chances to cover the same, possibly faulty, instruction. The MTK-QS approach aims to improve the effectiveness of the schedule by introducing an equivalence relation over test cases,and by rearranging the test case ordering according to the quotient set of this equivalence relation. More precisely, MTK-QS considers the quotient set induced by the equivalence relation of test cases having the same scores, and it rearranges the test suite by selecting, in turn, test cases from each equivalence class.

Studies on RTP presented in the literature typically involve an empirical evaluation of proposed novel approaches on a set of benchmark projects. Nevertheless, finding a suitable dataset is a nontrivial task, as authors typically provide only the data needed to replicate their proposed techniques. Furthermore, few studies evaluating fault detection metrics employ projects with real faults, and even fewer share these data. To overcome the problem to find a suitable dataset of subject projects, we constructed our dataset according to consolidated methodologies already applied in several prioritization studies.[14,15] Our dataset involves 25 different software versions among five real-world Java subject projects, with different characteristics both in their application domains and their sizes. As no real fault was available in the subject projects, we injected artificial faults through software mutations. We defined 53 experimental scenarios, that is, a pair of versions mirroring an evolutionary step in the subject

software life-cycle, and for each scenario, we generated 100 variants with different sets of faults. To assess the performance of a prioritization technique, we employed three fault-discovery metrics widely used in RTP studies, that is, average percentage of fault detection (*APFD*), percentage-to-first (*PTF*) and Percentage-to-Last (*PTF*) metrics. We also performed a time analysis of the proposed techniques, to assess their feasibility in real RTP scenarios.

The main contributions of this study are s follows:

- We propose MTK, an original approach to test case prioritization, capable of taking into account structural code changes to generate the permutation. To our knowledge, this is the first study in RTP leveraging a finer-grained measure of changes using TKs.
- We also design MTK-QS, an evolution of MTK which defines a strategy to enhance the rate of coverage of changed code, leveraging equivalence classes on the entity of the evaluated structural changes.
- We performed an empirical evaluation of both techniques on real-world software artifacts and compared the obtained results with well-known and widely used baseline techniques.
- We analyzed the computational times taken by the proposed techniques and compared them with the execution time of the baseline to define the practicability of our approaches.

The remainder of the paper is organized as follows: Section 2 describes the state-of-the-art approaches in RTP. Section 3 gives background details on TK functions and how we used them together with ASTs to evaluate source code similarity. Section 4 defines the details of the approaches we propose. Section 5 delineates the empirical methodology we followed to perform our experiments, while Section 6 shows the results of our evaluation and their analysis. Section 7 delineates the possible threats mining the validity to the generalizability of our study, and Section 8 synthesizes our findings and defines future research lines.

2 | BACKGROUND AND RELATED WORKS

Regression testing is a practice widely used in industry for the evolutionary maintenance phase of software, aimed at verifying that recent updates to the code-base have not degraded previously validated software features. It is however a resource-consuming practice, as it can typically account for up to 60% of the cost of maintenance.[16–18]

Historically, a naïve and sufficient approach to regression testing consisted of simply re-executing the entire software test suite to detect the failures of test cases after each update (i.e., the *retest-all* approach). With the evolution of software developmental methodologies, however, this approach is unfeasible when resource constraints are placed on the software testing phase. To cope with the issue, three main classes of different approaches have been studied,[1] *test suite minimization*, *test suite selection*, and *RTP*. Test suite minimization is a process aimed at identifying and then eliminating obsolete or redundant test cases from the test suite. This way, it is possible to reduce the amount of time and resources needed to run the test suite while hopefully preserving its fault-revealing capability.[2] Similarly, test case selection aims at selecting a meaningful subset of test cases to execute but typically does so by choosing tests that can be good trade-offs between code coverage and test costs.[19] Finally, RTP seeks a *permutation* of test cases in the test suite which maximizes desirable properties, such as fault detection rate. Unlike the former approaches, RTP preserves the integrity of the test suite, and test cases are executed according to the produced permutation while there still are available resources, maximizing the effectiveness of testing activities.[6]

The RTP problem can be formalized as follows.[20] Let $\mathcal{TS}$ be a test suite, $P(\mathcal{TS})$ be the set of permutations of $\mathcal{TS}$, and f be a target function which assigns a real number, representing an *award value* for the desirable property, to each permutation in $P(\mathcal{TS})$. The test case prioritization problem amounts to finding the permutation $T \in P(\mathcal{TS})$ that maximizes f, that is,

$$T = \underset{T' \in P(\mathcal{TS})}{\arg\max} (f(T')) \tag{1}$$

The prominent desirable property to maximize should be early fault detection, so that a large percentage of faults is detected even in case of abrupt termination of the regression testing activity. Nevertheless, an optimal approach maximizing the rate of fault detection is not possible to realize, as it needs prior knowledge of which test case will fail before actually executing it. Thus, heuristic approaches exploiting surrogate properties, which try to estimate the fault detection rate, are in practice employed as alternatives.[1,6] These surrogate properties are typically defined based on information such as test coverage, models of the software under test, history of the project, and requirements.

Several studies in literature leverage test coverage information to schedule the execution of test cases,[21] assuming that maximizing the rate of source code coverage can increase the chance of revealing faults. Two main *greedy* strategies are typically employed in coverage-based test case prioritization: the *total* and the *additional* strategies.[22] The former prioritizes test cases by sorting them in decreasing order of *total* number of source code units they cover, while the latter prioritizes test cases in decreasing order of code units that are covered by a given test but were

not yet covered by the previously executed test cases. In the literature, the coverage information has been used at different levels of granularity, for example, statement coverage,[23] branch coverage,[24] modified conditions/decision coverage,[25] and function/method coverage.[26] More recently, a hybrid coverage criterion was proposed in literature,[27] namely, the *code combinations coverage*, which combines traditional structural code unit coverage with the concept of combinatorial coverage. Some other studies[28,29] have also proposed a combination of standard test coverage with code change (i.e., the *code churn*) information. In another approach, Beszédes et al[28] considered procedure-level test coverage information and devised a strategy focused on covering the procedures that were affected by changes in the new version of the software.

To alleviate the nontrivial costs of collecting test coverage information,[6] some researchers have proposed model-based prioritization approaches.[30] These approaches are based on the creation of a model, that is, a simplified abstraction of the system. The execution of the suite is simulated on the model to gain useful information, such as application state transitions or coverage information. The main benefit of these techniques is a reduction of the overhead to gather such information due to this simplification. Korel et al[30] designed a prioritization strategy which relies on a *finite-state machine* model of the system, assessing higher priority to tests covering state transitions in parts of the model which were updated upon the new release. Although these approaches have been found to be promising also in improving early fault detection,[31] the cost to build and update a suitable underlying model, on which these techniques leverage, could not be affordable in all scenarios.

History-based prioritization techniques leverage information of the project previous evolution and testing phases, trying to predict which test case can be failing in the new version in release. Typically, these approaches exploit the software history to obtain information on test cases which failed in previous executions.[32,33] Lin et al[34] 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[35] takes into account the fact that test cases evolve as well, by defining a class of prioritization strategies that rank tests based on their similarity to 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 and retained (e.g., through the analysis of 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 test modification, addition, and removal can severely limit the historical information of test cases.[6]

Test case similarity-based techniques define a similarity measure between test cases and re-order the suite so that two subsequent test cases are as dissimilar as possible. Traditionally, these approaches are based on the grouping of test cases into clusters,[36–39] according to the similarity of dynamic traces derived from the execution of test cases. Similarity has also been exploited through the usage of the Jaccard distance between the set of covered code units of the test cases, scoring test case pairs, and ranking the cases accordingly.[40]

Search-based techniques introduce meta-heuristic search algorithms in the domain of test case prioritization. Li et al[41] were among the first to explore these approaches, by comparing, in their empirical study, some greedy prioritization techniques with hill climbing and genetic algorithm-based approaches. Genetic algorithms were also employed to search for an optimal ordering of test cases in another work,[42] which also leveraged history-based information. Other studies in the field employed nature-inspired meta-heuristics, such as *ant colony*,[43] *bee algorithm*,[44] *fish school*[45] and *particle swarm*,[46] to search for a solution which maximizes different properties, for example, historical metrics or rate of coverage. More recently, prioritization strategies employing machine learning techniques have been investigated. The majority of these techniques is based on *supervised* or *reinforcement* learning and are often realized to produce a prediction model which can rank test cases using different software metrics and historical data as features.[47,48] Even if these 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.[49]

To the best of our knowledge, existing prioritization techniques use a limited amount of information on code changes (e.g., consider only aggregated metrics related to changes) or do not exploit this information at all. Indeed, existing works that take into account code change information[29] consider only the extent of code changes (i.e., which structural code unit was affected by changes and which did not), completely disregarding the impact of said changes. With these techniques, a method which is changed because of a simple refactoring operation consisting of the renaming of a local variable is treated in the same manner as a method that was changed by completely rewriting its inner control flow. It is reasonable to assume, however, that the latter change has a higher risk of introducing faults than the former. Due to this motivation, our work aims at capturing these kinds of differences by focusing on the structural extent of changes between the two software versions and leveraging this finer-grained information to enhance the effectiveness of prioritization.

The impact of code changes has been extensively studied in the literature, particularly in the domain of *Software Change Impact Analysis*.[50] Several proposed techniques in this field leverage different structural models to predict the impact of code changes on an evolving software, often involving graph or tree-based structures.[8,51] ASTs are among the most widely used models in literature[52,53] and are often used in *Software Evolution Understanding* and *Code Summarization* domains. More recently, deep neural network approaches converting the changes in the code to an automated *embeddings* representation have been proposed[54] and have been applied to task as log message generation and code patch prediction. The semantic of the code has also been exploited using deep belief networks,[9] which were trained to learn *semantic vectors* from the source code AST representation for the task of predicting defect in changing software.

However, these techniques are applied to tasks that do not need to withstand the strict time constraints which characterize RTP and therefore they can not be suitable to this context. To this end, we designed our approaches on a lightweight framework possibly introducing low overhead in the steps to perform prioritization tasks.

3 | TKS FOR SOURCE CODE SIMILARITY

This section provides background notions on *TK* functions and describes how they can be used to quantify the extent of changes occurring between two versions of a software, leveraging an *AST* representation of source code.

3.1 | TK functions

Kernel functions have been extensively studied in *machine learning* and are employed to evaluate the similarity between objects by mapping their data points to a typically higher dimensional space. *Convolution kernels*[55] are a family of kernel functions which are used to evaluate the similarity between discrete structures, such as sequences or graphs. Convolution kernels evaluate similarity by summing the contribution of *fragments*, which are specific types of substructures embedded in the primary structure. Different types of fragments lead to different convolution kernel functions.

Among convolution kernels, *TKs* are kernels specifically designed for evaluating the similarity of tree-based structures.[56] TKs have been successfully and widely employed in many fields of computer science (e.g., *Natural Language Processing*[57] and *Software clone detection*[12]). Given two tree structures T_1 and T_2, the similarity score evaluated by a TK to the trees is expressed as[57]

$$K(T_1,T_2)=\sum_{n_1\in T_1}\sum_{n_2\in T_2}\Delta(n_1,n_2) \tag{2}$$

where n_1 and n_2 iterate on all nodes of the first tree T_1 and the second tree T_2, respectively. The Δ function defines the contribution to the overall similarity by the tree fragments rooted in nodes n_1 and n_2 and defines the particular type of TK in the function family. The similarity is a real number in the range $[0,+\infty[$. It is possible to normalize the resulting value in the range $[0,1]$, according to the following formula[57]:

$$K'(T_1,T_2)=\frac{K(T_1,T_2)}{\sqrt{K(T_1,T_1)\cdot K(T_2,T_2)}} \tag{3}$$

where K' is the *normalized kernel* function.

Different types of Δ functions have been proposed in the literature, defining several TK functions. The most popular and commonly used are *Sub-TK*, *Subset-TK*, and *Partial-TK*[58] *(PTK)*.

Sub-TK considers whole subtrees as fragments, tat is, a subtree rooted in node *n* includes all of its descendants up to the leaves. The Δ function of *sub-TK* assigns a score of 1 if the two subtrees rooted in nodes n_1 and n_2 are equal; that is, they are roots of two subtrees with the same structure and whose nodes have the same labels or 0 otherwise. The resulting sub-TK function is therefore a count of the number of subtrees the primary trees have in common. Figure 1B shows some subtree fragments of the tree depicted in Figure 1A.

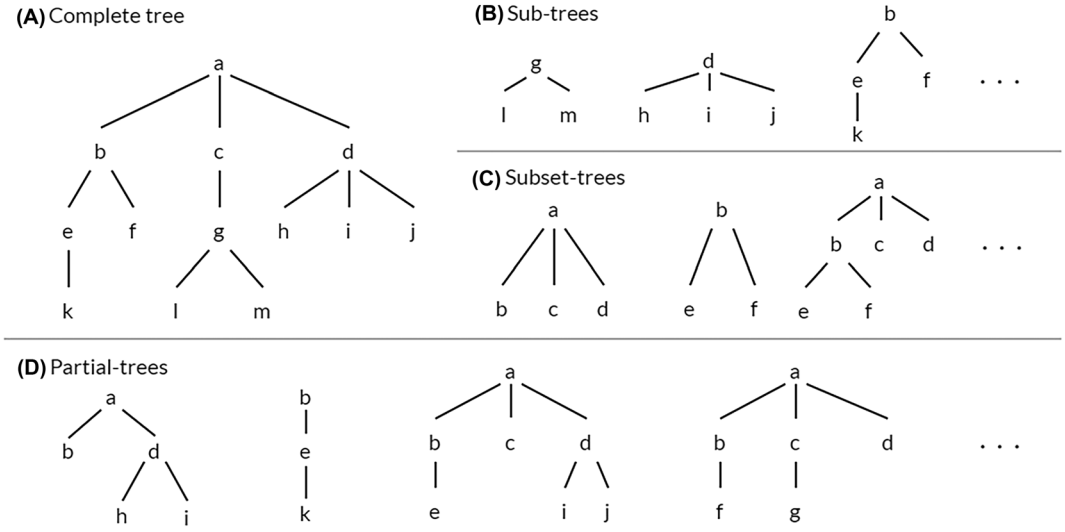

FIGURE 1 An example of different types of fragments for a tree structure: (A) the original tree, (B) some subtrees, (C) some of its subset-trees, and (D) some partial-trees.

Subset-TK, on the other hand, considers *subsets* of a tree as fragments. A *subset* of a tree rooted in a node n consists in the node n itself and some of its descendants, at any depth, with the only constraint that if a node is included in the subset-tree, it should include all of its siblings. Again, the specific Δ function for the *Subset-TK* assesses a score of 1 if two subset-trees are equal, 0 otherwise. This is actually a count of the number of subset-tree fragments the two trees share. Figure 1C depicts some subset-tree fragments.

Partial-TK relaxes the constraint of the Subset-TK, allowing to consider also subsequences of children of a node. In this case, given a tree node n, a *partial-tree* fragment is a subtree that includes any descendant node at any depth, with any sequence of children of any node. Some examples of partial trees are shown in Figure 1D. As the number of partial trees typically has a higher order of magnitude with respect to the node of the tree, it is possible to tune the *PTK* in order to progressively reduce the contribution of deeper levels when evaluating the similarity through a *decay factor*. Furthermore, an additional parameter is used to decrease both the impact of longer subsequences of child nodes and to penalize subsequences that present gaps with respect to the original sequence.

The contribution of two tree nodes n_1 and n_2 evaluated by the *PTK* Δ function to the overall similarity is equal to 0 if the two nodes have different labels; otherwise, it is a real value in $]0,1]$, depending on these tunable parameters. The Δ function for *Partial-TK* can be recursively defined as follows[58]:

$$\Delta(n_1,n_2)=\begin{cases}0 & \text{if } label(n_1)\neq label(n_2)\\ \mu\left(\lambda^2+\sum_{J_1,J_2:l(J_1)=l(J_2)}\lambda^{d(J_1)+d(J_2)}\cdot\prod_{i=1}^{l(J_1)}\Delta(c_{n_1}[J_{1i}],c_{n_2}[J_{2i}])\right) & \text{otherwise}\end{cases} \tag{4}$$

where J_1 and J_2 are sequences of indices for the direct children of nodes n_1 and n_2, respectively, while $c_n[i]$ refers to the i-th child of node n. The function l defines the length of a sequence of indices and the function d counts the gaps in a sequence. Parameter $\mu\in[0,1]$ is the decay factor that penalizes the contribution of deeper levels in the tree, and the $\lambda\in[0,1]$ parameter penalizes the contribution of longer sequences in the calculation. The overall calculation of the *Partial-TK* using the formulas in 2 and 4 can be implemented with a dynamic programming approach[57] with a time complexity of $O(\rho^3|T_1|\cdot|T_2|)$, where $|T|$ denotes the number of nodes of a tree and ρ is the maximum branching factor between T_1 and T_2.

3.2 | The AST representation of source code

Source code is written as a sequence of characters, collectively composing its textual representation. Although this representation is easily understandable by humans, it can be hard to automatically highlight the structural relations between sections of code, for example, statements in blocks or expressions in statements. These structural relations are typical of all modern high-level programming languages and can be defined by tree-based models, which naturally arise from source code and which are employed in several tasks involving source code analysis. The paramount tree-based representations of source code are *syntax trees*, which model source code with *ordered labeled tree* structures, drawing particular attention to the hierarchy of sections and fragments in the code and representing it as parent-child relations (e.g., a `for` loop is modeled as the parent node of all the statement nodes in its body). Children of a higher-level construct node are modeled as an ordered sequence, according to the order of their appearance in the source code. Tree structures are typically used as intermediate structures during the code compilation phase, the most widely employed being *parse trees* and *ASTs*. Parse trees are a direct transformation of a programming language grammar rules to a tree-based model, including language-specific symbols such as semicolons, brackets, or keywords.[59] This kind of modeling suffers from two main drawbacks: (i) as a mere tree-structured representation of grammar rules, parse trees are highly related to the specific language and their structure can widely vary between different programming languages, and (ii) they present a high-level of redundancy which causes the growth of deep and width of trees even for simple statements.

ASTs are not tightly coupled to grammar rules as parse trees, and their inner nodes typically have more meaningful labels representing the particular type of construct it models. Leaf nodes model *tokens* in the source code, for example, the name of a variable or a literal value, which a leaf node is labeled with. Figure 2 shows an example of an *AST* representation for a simple Java method. In the figure, the *body* block of the method has two children, one related to the *If* block and the second representing the *Return* statement. Note that this organization of the children sequence resembles the order in which the statements appear in the source code. Each block has its subtree with simpler constructs, eventually descending to leaves which are labeled with source code tokens.

AST models provide a more concise representation of source code structure, reducing structural redundancies related to parse tree, both increasing the meaningfulness of the representation and pruning nodes related to language-specific symbols and keywords. Moreover, since AST representations are not tightly defined by grammar rules of the programming language, it is possible to define different AST representations of source code specifically tailored towards particular tasks, for example, by highlighting specific features which are significant for the task at hand. Furthermore, they can be used on programming language-agnostic tasks and are thus more generally applicable as the underlying model for many applications.

For these reasons, in order to evaluate the structural similarity between source code, we employed an *AST* representation as the underlying model.

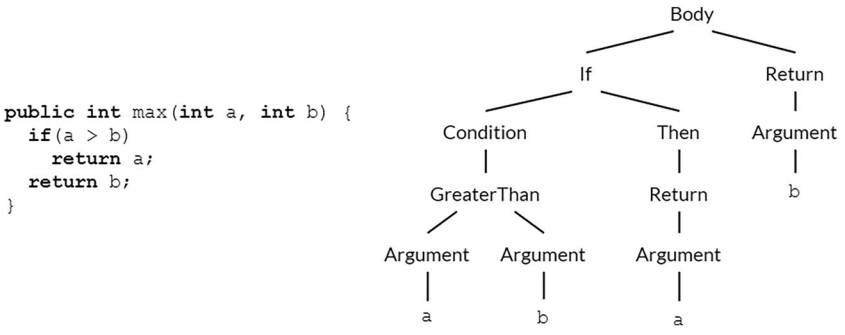

FIGURE 2 An abstract syntax tree (*AST*) representation for a simple Java method that returns the maximum between two integers.

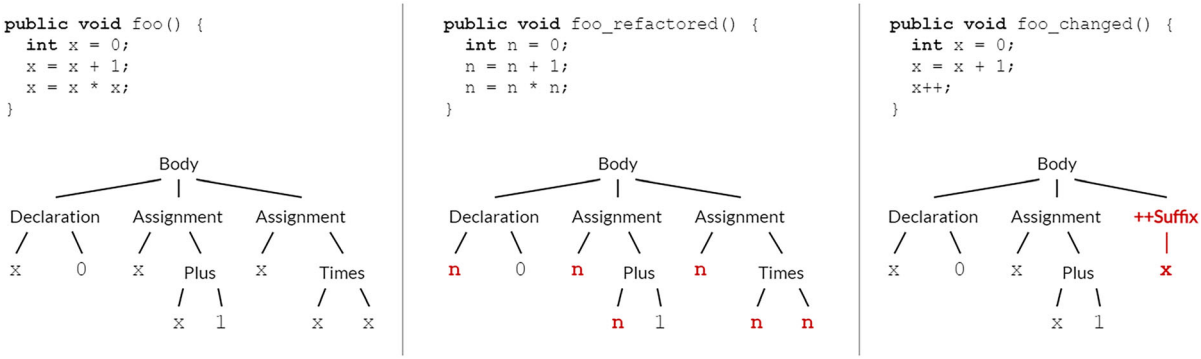

FIGURE 3 Different changes in abstract syntax tree (AST) representations according to changes in the source code. The original `foo` method (left) is refactored by renaming its variable (center) and with the removal of a statement and the addition of a new one (right). Changed nodes are highlighted.

3.3 | Measuring the extent of changes in source code with TKs

To perform a quantitative analysis of the similarity between two source code ASTs, after a preliminary investigation, we chose *Partial-TK (PTK)*. The main motivation for this choice is that we found PTKs to be more suitable to highlight common operations to the code base when software evolves. In fact, changes occurring in a developmental step involve the addition or the removal of expressions, statements, or blocks, which reflect in the addition or removal of nodes in the sequence of children in the AST. As PTK takes into account also partial-trees, it can produce a more refined similarity value for these kinds of operations. Furthermore, there are software evolutionary practices in which a high number of changes reflects poor or no semantical modifications at all, such as refactoring (e.g., the simple renaming of a variable in a method). In such cases, PTK can be more aware of these changes than sub-TK or subset-TK, due to the different types of tree fragments it considers when evaluating the similarity. We performed several experiments to assess this phenomenon and to qualitatively find suitable values for the parameters of the partial-TK. To this end, we found that $\mu = 0.4$ and $\lambda = 0.7$ generally resulted in accurate evaluations of structural similarity.

As an example, consider the modification of the `foo` method in Figure 3. The `foo_refactored` method presents a renaming of the local variable `x` to `n`. A naive textual approach will mark every line as changed, as they are all different with respect to the original method. As it can be seen by the AST representation, the renaming only affects the leaves (i.e., lexemes) related to the former `x` variable, while the tree structure remains untouched. An application of the sub-TK to the original and refactored method, however, assesses a rather low similarity score of 38%. This is due to the fact that almost every subtree fragment is changed between the two methods, and therefore, the count in Equation (2) is small. On the contrary, PTK results in a 93% similarity between the two methods, as there is a 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 just this line as changed. The changes in the method's AST are however more pronounced, as a whole subtree is prominently different while the others remain unmodified. In this case, the sub-TK provides a normalized similarity score of 91%, as just one subtree fragment does not contribute to the total. The PTK produces instead a score of 73% due to the higher number of fragments which mismatch and the evaluated similarity results

lower. As shown by the example, we consider that partial trees can produce a more refined score both with respect to a naive textual similarity and to other kinds of fragments.

The time complexity of PTK is linear in the number of nodes of both trees but cubic in the branching factor ρ. This means that similarity for trees with a high branching factor can be heavier to evaluate with respect to trees which extends more in depth than in width. As a consequence, the evaluation of the PTK consumes more time on ASTs which represent methods with long sequences of statements at the same nesting depth. On the other hand, methods with several levels of nested blocks have a lower impact on the evaluation time.

4 | TKS FOR RTP

During the evolution of software due to maintenance activities involved in its life cycle, the implementation of different modules and packages is updated by developers, in order to correct unintended behaviors, add new functionalities or optimize previously implemented functions. These activities typically lead to changes in the source code (i.e., the *code churn*) of modules which were already validated and often involve deep alterations of the code units structure. Changes in the source code structure can easily be prone to errors and mistakes, thus invalidating prior working functionalities. Due to this reason, we deem that code units which were subjected to greater changes in their structure should be the main focus of regression testing activity.

To this end, we propose two novel approaches to prioritize test cases. The first approach is based on the exploitation of the structural similarity between the source code of two software versions to evaluate the *code churn* using *TK* functions and then scoring test cases accordingly. The second approach is an evolution of the first one and aims to enhance the diversification of coverage on changed methods, in order to accelerate the exercise of different changed code units. This approach involves the partitioning of the reordered test suite in equivalence classes, basing on the changes evaluated by TKs. The test suite is then re-arranged by choosing test cases from these equivalence classes in a round-robin fashion. This approach, called *quotient set prioritization*, can be applied as a complementary step to any prioritization approach which defines a scoring measure for test cases. An example of the steps involved in the application of both techniques is shown in Figure 4.

4.1 | Tree kernel-based prioritization

The first prioritization approach we propose leverages *code churn* information along with past *coverage reports* to prioritize test cases in a test suite. We adopted *TK* functions to quantify changes between two versions of a software project, in a way that takes into account not only the number of changes but also their impact on the syntactic structure of the program. The measure of changes is then joined with coverage information to assign a *score* to each test case based on the number of changed methods the test covers, weighted by the dissimilarity of the method

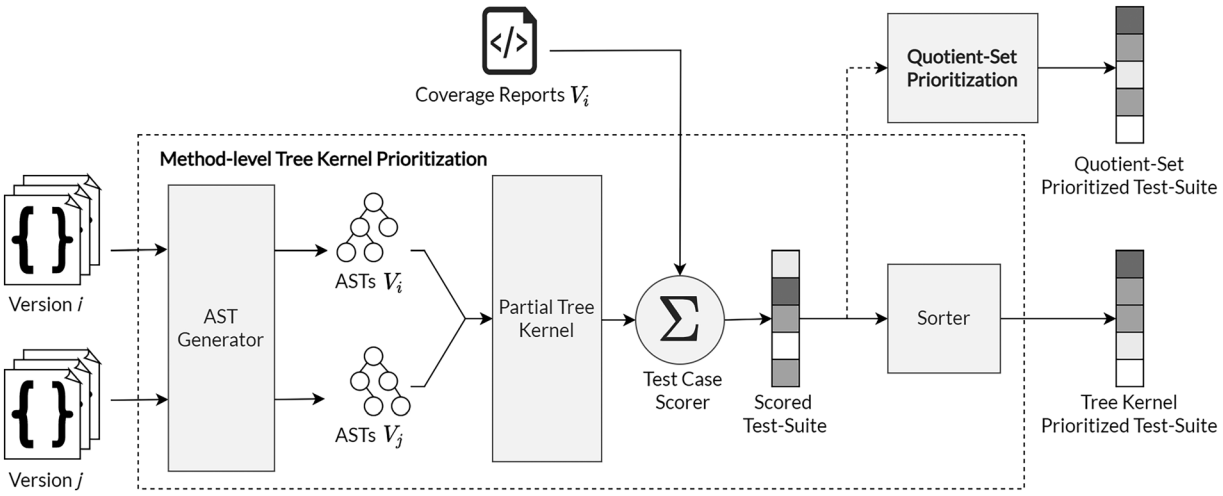

FIGURE 4 Execution steps for method-level tree kernel prioritization and quotient set prioritization. The techniques take as input the previous software version V_i and the new release V_j, along with the coverage reports at a method-level granularity of V_i. Scores assigned to test cases obtained by the *Test Case Scorer* module are passed to the quotient set prioritization to build the equivalence classes and are re-ordered accordingly. The two outputs are independent and refer to the application of the two proposed techniques.

between the two versions. The suite is then sorted in descending order using the evaluated scores, with the rationale that code changes are prone to faults and test cases covering a higher number of structural changes should have a higher probability to exhibit faults.

Given an evolutionary step of the software from version V_i to version V_j, we extract the *AST* representation from the source code of each method of both versions. Therefore, we construct the matching $\mathcal{M}$ as the set of pairs (m_p, m_c) of methods which were changed between these two versions, where m_p represents the method in the *previous* version V_i and m_c is the method in the *current*, new version V_j. For our purposes, two methods $m_p \in V_i$ 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 m_c in the current version satisfies this constraint, the method m_p remains unmatched. Note that every method is matched at most once, as we use the fully qualified name of the method along with its signature to identify the pairs, which create a unique identifier for the method when they are concatenated together.

The pseudo-code for the evaluation of the method matching $\mathcal{M}$ is presented in Algorithm 1. The algorithm creates a map of methods in the previous version V_i (lines 1–4). The key of the map is the *qualified signature* of methods, that is, a string with the full classpath of the method with package, classes, method name, and the type of all of its arguments. Then, for each method m_c in the current version V_j, the algorithm checks whether the qualified signature of m_c is contained in the map of the previous method. On a successful check, the previous method m_p is retrieved from the map and the pair (m_p, m_c) is added to the matching $\mathcal{M}$, on lines 7–9. Eventually, the method matching $\mathcal{M}$ is returned in line 12.

Algorithm 1 Creation of the matching between methods of two versions.
Input: $\mathbf{V_i}$: previous version sources; $\mathbf{V_j}$: current version sources
Output: $\mathcal{M}$: a set of 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
```

For a pair of matched method $(m_p, m_c) \in \mathcal{M}$, we evaluate the structural similarity $s(m_p, m_c)$ using the *normalized partial tree kernel* (*PTK*) function, defined in Section 3.1, on their respective ASTs. Higher values of similarity on the ASTs are related to a higher structural similarity between the source code of the methods. A score of 1 represents the complete equality between the trees, that is, the method is not changed in the evolutionary step, while a score close to 0 means that a great extents of modifications have been performed on the method. The absence of modifications in the body of a method can be assessed also with a simple textual comparison of their source code, directly. It is therefore possible to compare the source code of the two versions of the method *before* the evaluation of their AST similarity with PTK. If the textual representation has been changed, we simply assess a similarity score of 1 and proceed to the next method. Otherwise, the PTK function is normally applied to the ASTs. This consideration speeds up the similarity evaluation process, and it is of particular benefit when the number of changed methods is small compared to the size of the project.

At the end of the process, each method in version V_j is labeled with the evaluated score complemented to 1, that is $1 - s(m_p, m_c)$, which reflects the entity of structural changes, that is, the *dissimilarity*, of the method since version V_i.

We note that, in our approach, methods that have been renamed between versions V_i and V_j are not matched. This choice has a twofold rationale. On the one hand, if the method is simply renamed between the two versions, with no changes to its source code, the proposed TK-based techniques would assign a dissimilarity of 0 to it anyway, effectively ignoring it during prioritization. On the other hand, in cases where there are also changes to the source code, it is not straightforward to discriminate between a newly added method and a method that was both renamed and significantly modified. Doing so would require setting some arbitrary similarity threshold to determine whether a method is a renamed and evolved version of a previously existing method or an entirely new method. Defining such a heuristic approach for matching and demonstrating its validity requires a dedicated and thorough investigation of its own and is out of the scope of this paper.

To join test cases and churn information obtained through PTK, we use *test coverage* information at method level for the previous version V_i in the software incremental step. We exclude removed and added test cases from the prioritization. In fact, removed test cases cannot be executed anymore and thus are not included in the test suite permutation for the current version, while freshly added tests have not been executed in the previous version and possess no coverage data, so it is not possible to know which methods in the source code they exercise. Given a test

T in a test suite $\mathcal{TS}$, we denote with $Cov(T)$ the set of all methods m in version V_i which have been matched with a method m_c in version V_j and with $\mathcal{M}(m_p) = m_c$ the mapping of the method m to its paired method m'. Thus, the *score* of test T is evaluated as follows:

$$score(T) = \sum_{m_p \in Cov(T)} (1 - s(m_p, \mathcal{M}(m_p))) \tag{5}$$

Eventually, the test suite is sorted by the evaluated scores in descending order. We called this approach *method-level tree kernel prioritization* (*MTK*) and its *pseudo-code* is presented in Algorithm 2. The first line of the algorithm calls the auxiliary unction *createMethodMatching*, which executes the steps in Algorithm 1 to create the set of matching methods $\mathcal{M}$. Lines 2–8 evaluate similarity between the pairs of matched methods in $\mathcal{M}$. In particular, if the textual representation of the methods body is different (lines 3 and 4), the similarity between the methods ASTs is evaluated using the *normalizedPTK* function. The function is the implementation of the partial TK function defined in Equation (4), and its value is normalized according to Equation (4). For this function, we relied on the dynamic programming implementation available in the *KeLP*[60] framework*. If the text of the two methods is equal, the similarity is directly set to 1, and the TK function is not applied. 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 $\mathcal{M}$. 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 and 18).

Algorithm 2 Test case prioritization based on tree kernel.

Input: $\mathcal{TS}$: test suite for current version, $\mathbf{V_i}$: previous version sources, $\mathbf{V_j}$: current version sources, **Cov**: method-level coverage information for each test case

Output: $\mathcal{TS}'$: 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:          s[m_p, m_c] ← normalizedPTK(m_p.ast, m_c.ast)
5:      else
6:          s[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 ∈ Cov(T) do
12:         if ∃m_c : (m_p, m_c) ∈ M then
13:             T.score = T.score +(1 − s[m_p, m_c])
14:         end if
15:     end for
16: end for
17: TS' ← sortByScoreDescending(TS)
18: return TS'
```

4.2 | Quotient set prioritization

As it can be noted from Equation (5), 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, as it implies that two test cases cover different methods which were subjected to identical structural changes. Therefore, with the exception of extraordinary cases, two test cases with the same score cover the exact set of changed methods in the vast majority of practical scenarios.

The MTK approach defined in Section 4.1 orders the test suite according to *score* function values. This means that all the test cases which have 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 same sections of changed code, degrading the rate of faults discovered. In fact, faults introduced in other changed methods will be stressed by test cases which have been postponed in the ordering due to a lower assigned score. The main reason behind this

*https://www.kelp-ml.org/?page_id=728, visited on 30/12/2023.

behavior is that the score function is *local* to a single test case and does not consider which other test cases could be executed before it. In the worst-case scenario, this behavior leads to many undetected faults when the test suite execution is interrupted due to meeting the constraint of the testing phase.

Algorithm 3 Quotient set prioritization.
Input: $\mathcal{TS}_{/\equiv}$: quotient set containing equivalence classes of test cases with the same scores
Output: $\mathcal{TS}'$: 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 TS_/≡.remove([T]_≡)
11:         end if
12:     end for
13: end while
14: TS'.appendAll([z]_≡)
15: return TS'
```

For this reason, we evolved our MTK prioritization approach to include also information related to previously scheduled test cases in a permutation. It is designed to better diversify the coverage of changes for the test cases leveraging the score values evaluated by TK, with the intuition that the same scores will result in 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 put firsts. Finally, the prioritized test suite is organized by suitably picking a test case from each equivalence class in the given decreasing score order, following a round-robin policy until all test cases have been arranged. This approach, namely, *quotient set prioritization* employing *method-level tree kernel* (*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.

Given a test suite $\mathcal{TS}$, we define an *equivalence relation* $\equiv$ on it using the scoring function in Equation (5). For each test cases $T_1, T_2 \in \mathcal{TS}$, we say that $T_1 \equiv T_2$ iff $score(T_1) = score(T_2)$. It is easy to prove that $\equiv$ is an *equivalence relation*. We can thus partition the test suite $\mathcal{TS}$ in the *quotient set* $\mathcal{TS}_{/\equiv}$ as the set of *equivalence classes* of test cases with the same *score*. Note that it is possible to use any function which scores test cases in place of Equation (5) so that different test equivalence criteria can be evaluated.

Exploiting the quotient set $\mathcal{TS}_{/\equiv}$, our approach consists of re-ordering the test suite by picking test cases from these equivalence classes in a round-robin fashion, starting from those which have a higher score. The intuition is that this kind of re-ordering allows one to achieve a faster and larger coverage rate of methods which have been changed between the two software versions.

More in detail, we sort equivalence classes in $\mathcal{TS}_{/\equiv}$ by their scores in descending order. Then we choose a single test case from the equivalence class with the highest score and put it in the first position in the prioritized test suite, removing it from the class to which it belonged. We then select, in turn, a test case for all the subsequent equivalence classes, appending each test case after the others which were already placed in the prioritized test suite. Upon reaching the last equivalence class, the selection process resumes from the first one. We repeat this round-robin selection process until all test cases have been picked. Notice that, since all the test cases which cover no changed method have a score equal to zero, we exclude their equivalence class from the round-robin and simply append all these tests at the end of the prioritized test suite.

When selecting a test case from an equivalence class, several criteria can be chosen, for instance, by picking randomly a test in the class. For our purpose, we employed a *total coverage* criterion to select the test case to pick inside a class. This criterion chooses the test which covers the highest number of methods, both changed and unchanged. This has the goal of increasing the rate of total method coverage in the re-ordered test suite.

The pseudo-code of *quotient set prioritization* is shown in Algorithm 3. The first statement removes the equivalence class $[z]_{\equiv}$ from the quotient set $\mathcal{TS}_{/\equiv}$, as it will be appended after all other test cases have been inserted in the prioritized suite. Lines 2 and 3 implement the total coverage selection criterion, by sorting each equivalence class in descending order by method coverage. This allows picking test cases with higher method coverage inside each class before the other with the same score. The loop in lines 6-13 is the core of the prioritization and runs until no more test cases should be added by checking that 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 and 10 manages to remove empty equivalence classes from the quotient set, in order to eventually meet the `while` loop termination condition.

We designed the *quotient set* approach to cope with the redundancy issue of the *MTK* technique. The QS approach, however, relies only on the score assigned to test cases, with no constraint on the specific scoring function. This means that it is possible to use QS with every prioritization technique which ranks the test cases by assigning them a score. In this paper, we also apply the QS approach after another change-based prioritization technique, the *diff prioritization*, which scores test cases according to the number of changed methods they cover.

5 | EXPERIMENTAL DESIGN

Our study focuses on the impact of structural changes in the code on test case prioritization. To evaluate the structural similarity of methods which have changed between two software versions, we employed TKs and then scored test cases using the scoring function defined in Equation (5). The test cases are then sorted in descending score order to produce the prioritized suite based solely on TKs. To apply the quotient set prioritization, the test cases are partitioned in a quotient set according to their score and then permuted in an order which can better distribute code coverage.

The goal of our study is to assess the performance of TK prioritization and quotient set prioritization techniques compared with other baseline approaches, both aware and unaware of code changes. To this purpose, we performed the evaluation of the proposed techniques on a set of projects written in the Java programming language, in which we injected artificial faults, following methodologies widely used in literature.[14,15] As prioritization techniques are applied when a new version of the software is going to be released, we considered an evolutionary step in the software life cycle as the basis for each experiment. Hence, experiments are carried out starting from a pair of versions (V_i, V_j), with $i < j$, which represents an update to the code base from a prior version V_i to a more recent version V_j. Notice that, in general, V_j may not be the version immediately following V_i, that is, $j \geq i + 1$. We then executed our proposed prioritization techniques on each of such *experimental pairs*, along with other six baseline RTP approaches. A complete replication package is available on Zenodo[61] and contains both the dataset and the source code to reproduce all experiments used in this study.

5.1 | Dataset collection

In order to select the software projects and the related versions to use in our study, we took inspiration from the collection of Java projects considered in previous studies[14,15]: They are particularly heterogeneous, ranging from small projects with less than 3000 lines of code to large projects with over 100,000 lines. Some of the projects defined in the original studies were publicly available on the *GitHub* platform. However, while recollecting the data, we found that some projects had been moved or removed and could not be retrieved anymore. Other projects (e.g., *Metrics*) were not suitable for our evaluation due to a limited number of changes through its versions. For our experiments, we collected and employed five Java projects, along with 25 versions across these projects. Table 1 presents the details of benchmark projects.

Once the projects were retrieved from *GitHub*, we compiled them and ran their test suites to collect the coverage and test execution reports needed by our prioritization approaches. All projects used *Maven*[†] as a software management tool to ease the effort of executing the compilation and the testing phases. To collect coverage reports while executing the tests, we employed *OpenClover*[‡], an open-source coverage tool for Java, which was added as a dependency to each collected version. We also modified the tool to include *per-test* reports, as the original tool produced coverage reports aggregated at the whole test suite level. The build and test phases of each project have been executed on a *Docker*[§] container to recreate their original building environment as more as possible.

The collected projects exhibited no failures when the testing phase has been executed. Thus, in order to perform the evaluation of our techniques, we followed a widely accepted methodology[14,15] present in the literature. This methodology consists in adding artificial faults to each version through *software mutation*. Software mutation is a technique employed in several practical contexts to obtain adequacy metrics related to the quality of a test suite.[62] The mutation process is accomplished by changing parts of the source code of the software (e.g., substituting one arithmetic operator with another, changing the value of literals, or changing conditions in branching operators), and each such change is called a *code mutant*. We embedded these mutants in changed parts of the source code, which were already present between software versions in our project collection. Each injected mutation represents a fault in the considered evolutionary scenario. To this end, we assume that a test case reveals one or more faults if it covers the exact statements in which the mutants were injected.

To prepare the setting for our experiments, for each project, the source code of all versions in our dataset has been mutated, and the set of all code mutants has been put into a *mutant pool*. Many Java mutation frameworks produce mutants directly in the *bytecode*, and these mutants cannot be generally mapped to alterations of the original source code. Our choice in the mutation framework fell back to *Major Mutation Framework*,[63] which has the advantage of producing the code mutants directly mutating the original source code of the software.

[†]https://maven.apache.org, visited on 30/12/2023.
[‡]https://openclover.org, visited on 30/12/2023.
[§]https://www.docker.com, visited on 30/12/2023.

TABLE 1 Overview of projects used in our empirical evaluation.

Project	Versions	LoCs	Methods	Tests	Common	Changes	Description
AssertJ	6	60k	7985	5134	4994	62	A testing framework for Java
JOpt	4	7k	860	639	532	14	A Command-Line Interface parsing library
Joda-Time	4	160k	9412	4183	4045	28	A date and time processing library for Java
La4J	6	9k	992	383	328	38	A linear algebra framework for Java
Scribe	5	3k	344	74	61	12	A Java library implementing the *O-Auth2* protocol

Note: *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.

The collected mutant pool was then filtered by removing mutants which were not detected by any test case. The list of all types of mutant operators and their description is presented in Table 2. The mutation operators we employed are part of those provided by the *Major* framework[¶]. We chose these types of operators as they can both resemble programming errors and can alter the flow of instructions in the method, thus leading to unintended behaviors. The applied mutants replicate all the types of faults used in a previous study.[15]

For each pair (V_i, V_j) within the same project, we evaluated the set of changed methods between these versions to determine the code sections in which mutation faults could be embedded. From all the 68 possible version pairs, we discarded 15 of them as the number of changes between these versions was too small to have a reasonable number of points in which to inject the faults, leaving us with 53 version pairs. We subsequently created *variants* of the version V_j in each pair. To create a variant, we randomly injected five mutation faults in the methods which were changed from the previous version. We chose to use groups of five faults as prior research[15] stated that the number of mutations does not affect the evaluation metrics significantly. We, therefore, aligned our experiments with other studies in the literature which employ five mutation faults.[14,64,65]

More specifically, we selected a subset of five mutants from the mutant pool with a uniform probability distribution. The source code of the current version was then merged with the source code of the mutants in the set to produce a variant. To enhance the significance of our experiments, we constructed 100 variants for each one of the 53 pairs of version (V_i, V_j). These 100 variants simulate faults in different changed parts of the software.

5.2 | Empirical evaluation setting

Our experiments involve the evaluation of our prioritization approaches in software evolution scenarios. Each experiment is based on a pair of software versions (V_i, V_j), with i and j ranging from 1 to the index of the last version in the project and $i < j$. V_j is intended as the current (i.e., new) release of the software, and V_i is the previous version which is updated. We assume that coverage information, test execution reports, and full source code for V_i are available. The target of our experiments is to prioritize the test cases of the current version in the presence of faults. To do so, we execute our prioritization approaches on all variants of the current version related to the pair (V_i, V_j), obtaining 100 evaluations for each pair. Note that test cases newly added in version V_j are not considered in the prioritization, as no coverage information is available in V_i for them. Thus, we aim to prioritize only the *common* subset of the test suite between the two considered versions. We rely on the qualified name of a test case to determine if it is present in both versions.

We implemented MTK and MTK-QS prioritization techniques in Java. To extract the AST of source code, we employed the *GumTreeDiff* tool[66] and the TK implementation available in the *KeLP* framework,[60] which have been used in several previous studies in Natural Language Processing and Software Engineering.

For our experimental setting, we chose six baseline techniques to which our approaches are compared:

- *Untreated* (*NOP*): the test suite is not re-ordered at all, and its test cases are executed in the order defined by developers or by the test runner (e.g., *JUnit*).
- *Total* (*total*): re-orders the test suite according to the number of *covered methods* of each test case. The technique has been implemented *from scratch* based on its definition in literature.[23]
- *Differences priority* (*diff*): gives higher priority to those test cases which cover a greater number of changed methods.[29] This approach assigns a score equal to the number of changed methods to each test case and ranks the test suite accordingly. For this strategy, a method is considered changed if the text of at least one of its statements has been changed (ignoring changes in comments). It has been implemented through the

[¶]The complete list of mutation operators can be found at https://mutation-testing.org/docs.html#mutation-operators, accessed on 30/12/2023.

TABLE 2 Complete list of all mutant operators used in this study.

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.

git diff utility, which provides the differences between two *commits* at different levels of granularity. Information on the changed methods is joined with coverage reports, and the score of each test case is evaluated as the number of changed methods covered by the test.

- *Quotient set on differences* (*diff-QS*): a variant of the *quotient set* prioritization, defined in Section 4.2, which uses scores evaluated by the *diff* prioritization to build the quotient set. We implemented this strategy atop *diff* prioritization.
- *Genetic*: a genetic approach to RTP which maximizes *average percentage transitional coverage* (*APTC*) at method level, that is, the rate at which a permutation of the test suite covers methods. To implement the approach, we used a previous available implementation,[14] with some minimal modifications to allow its usage in our prioritization platform. When executing the algorithm, we set the population size and the number of generations both equal to 100.
- *Adaptive random testing max-min* (*ART*): an adaptive random technique which randomly selects the next test case among the ones maximizing the minimum distance to the already scheduled tests.[40] Distance between two test cases is evaluated as the Jaccard index between the sets of the covered methods of the respective test cases. As in the case of *Genetic*, we used an existing implementation[14] which we adapted to our framework.

We chose different types of baseline techniques according to whether or not they are aware of source code changes. The *total*, *Genetic* and *ART-MaxMin* techniques rely only on coverage reports and use no information of the *code churn*. We included them in the baselines to compare our proposed techniques against well-known standard approaches. The *diff* prioritization strategy considers *code churn* information in a coarser fashion than MTK, and it was chosen as a baseline to empirically verify if a finer evaluation of similarity can be justified. The other change-aware *diff-QS* technique is presented to analyze the performance of quotient set prioritization using another change-based scoring function. The *NOP* technique is actually not a prioritization approach, and it is included only to evaluate the effective benefits of applying any prioritization technique on the subjects.

To evaluate the performance of our approaches and assess a quantitative comparison with the baseline techniques, we employed the *average percentage of faults detected* (*APFD*) metric.[23] *APFD* is a widely used metric in *RTP* studies[14,40,67] and measures the fault-exposing effectiveness of a reordered test suite, intended as the rate at which test cases uncover faults. It is defined as follows:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{mn} + \frac{1}{2n} \tag{6}$$

where m denotes the number of faults in the software, n is the cardinality of the test suite and TF_i is the index of the test case, in the prioritized test suite, which is first to uncover the i-th fault. The last summation term in Equation (6) is a normalization factor to obtain values in the range]0,1[. The higher the APFD value of a re-ordered test suite, the quicker its execution can discover failing test cases and thus faults in the software.

The APFD metric gives insights into how quickly a permutation of the test suite discovers faults. However, it does not give intuitions on how many test cases should be executed to find a failure or to discover all faults introduced with an update to the codebase. For this reason, we also considered two additional metrics which are complementary to APFD, that is, PTF fault and *Percentage-to-Last* (PTL) fault metrics. The PTF metric

measures the fraction of the test suite which needs to be executed before it is possible to find a test case discovering a fault in the permutation, while PTL evaluates the fraction of tests that should be run before all faults are discovered.

Given a permutation $\mathcal{TS}$ of test cases in a test suite, the PTF metric is defined as follows:

$$PTF(\mathcal{TS}) = \frac{FTF}{|\mathcal{TS}|} \tag{7}$$

where *FTF* is the index of the first test case in the permutation which fails.

Similarly, the PTL metric can be written as follows:

$$PTL(\mathcal{TS}) = \frac{LTF}{|\mathcal{TS}|} \tag{8}$$

with *LTF* being the index of the test cases which is the first to exhibit the last faults, that is, the point in the test suite execution after which no other faults are left undiscovered. PTF and PTL range in the interval $]0,1]$, and, generally, for each permutation $\mathcal{TS}$, it is $PTF(\mathcal{TS}) \leq PTL(\mathcal{TS})$. The lower the value results for these metrics, the fewer test cases should be executed to detect the faults. Hence, PTF and PTL can suggest how many resources can be saved by a prioritization technique.

The metrics defined so far are related to the fault-revealing power of an RTP approach and can be generally used as proxy metrics to analyze the benefits of prioritization in relation to any kind of constrained resource. However, they do not give direct insights into the time saved for the testing phase by prioritization approaches. Thus, we introduce the *effective execution time* metric, which accumulates both the time spent executing the prioritization and the time needed by the permutation to discover all faults in the new release. Given a prioritization strategy *S* and its permutation $P = (t_1, t_2, ..., t_N)$, the *effective execution time* of *S* is as follows:

$$T_{\text{eff}}(S) = T_{\text{exec}}(S) + \sum_{i=1}^{l} T(t_i) \tag{9}$$

where $T_{\text{exec}}(S)$ is the execution time of the approach *S*, $T(t)$ is the execution time of test *t* and *l* is the index of the first test in the permutation discovering the last fault. A lower effective execution time for a technique is better, as it means more time saved by using it in the testing phase.

To investigate to what extent the number of changes affects the presented techniques, we also performed an analysis of the APFD values in relation to the number of changed methods across the dataset. We partitioned all the experimental pairs of versions into three groups, according to the number of changed methods between the versions: small (from 1 to 15 changed methods), medium (16 to 40) and large (more than 40 changed methods). This partitioning also produced a balanced distribution of experimental pairs in the groups.

To obtain insights also on the feasibility of using the proposed approaches in real-world scenarios, we investigated their execution time and compared them against the baseline techniques. To this end, we employed the *Java Microbenchmark Hareness*[#] (*JMH*), a popular Java framework which allows to define and execute benchmarks for Java code and computes several performance metrics. We used the *average time* metric reported from the framework. Furthermore, as considered projects have typically stable characteristics between all versions, we executed one experiment for each project in our dataset. For each project, we used the first and older available version V_1 of that project with respect to the last and newest one V_n. Since the execution time for the evaluation of the PTK function depends on the number of changed methods, the (V_1, V_n) pair will typically present the greatest number of changes and it could provide an upper bound on the time used by our strategies. The experiments were performed on a system with an Intel i7-4970 3.6GHz CPU and 24GB of RAM memory, using Microsoft Windows Server 2019 as the operating system.

6 | RESULTS

This section illustrates the results of our experimental assessment and their comparison to the baseline techniques. We aim to show the effectiveness of the proposed prioritization techniques on the dataset we collected, in terms of fault detection rate. We performed the comparison both according to the characteristics of subject projects and in relation to the number of changes between the pairs of versions considered in our experiments. For each project, we also compared the execution times of the proposed techniques with respect to the baselines on the scenario with the highest number of changes between the previous and the current versions available among experimental pairs. Furthermore, we analyzed the amount of time that can be saved by the proposed techniques, considering both the time needed to apply a specific approach and the

[#] https://github.com/openjdk/jmh, visited on 30/12/2023.

execution time of the fraction of test cases in the permutation exhibiting all the faults, normalized on the time of the whole test suite. This has the purpose to quantify the benefits of the techniques application, and to compare these results with all the baselines.

6.1 | Fault detection performance

To simulate different extents of changes for evolutionary steps in the software, we chose all possible pairs of versions (V_i, V_j) available in our dataset, where version V_i is chronologically older than V_j. This setting covers several evolutionary scenarios for all projects in the dataset, with regard to the number of changes between the older version V_i and the new release V_j. We executed the proposed prioritization techniques, along with the baseline techniques, on all such pairs of versions, considering all the variants with injected faults of the right version. Then, we evaluated the APFD, PTF, and PTL for each permutation produced by these techniques.

We also performed a *Wilcoxon–Mann–WhitneyU-test* on the results of each technique to assess whether statistically significant differences exist. To this purpose, we consider the following *null hypothesis*:

$\mathbf{H_0^{t_1,t_2,p}}$: 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.

The confidence limit, α, was set at 0.05, and as also done in other software engineering studies,[68,69] we applied the standard Bonferroni correction (α/K, where K is the number of hypotheses) when multiple hypotheses were tested, to mitigate the *multiple comparisons problem*.

The APFD results of the techniques are shown in Figure 5. For each considered pair of versions in each project, we aggregated the APFD values obtained by all techniques for all the variants, then we presented a box for both the baseline techniques and the proposed approaches. As it can be seen, any prioritization approach significantly improves the rate of fault detection with respect to the untreated test suite (*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 techniques exhibit an increased average rate of fault detection.

The APFD results for the MTK technique are generally lower than (or at most comparable with) the baselines for the majority of projects. The main reason for this behavior is the redundancy of scheduled test cases. In fact, several test cases, which cover the same changed methods and

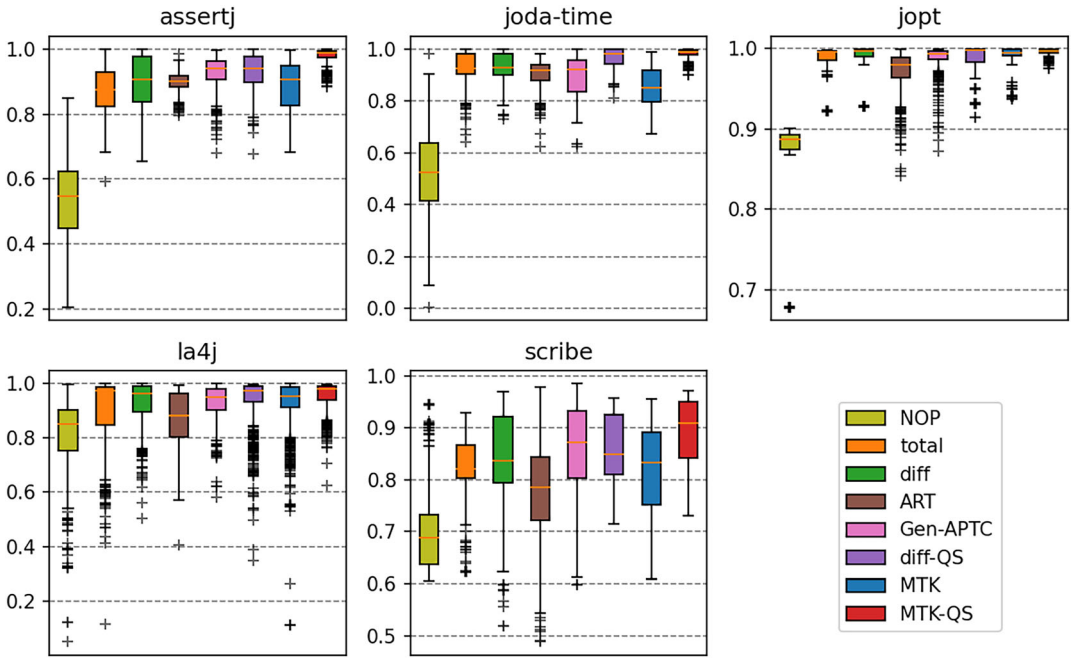

FIGURE 5 Box-plots of resulting average percentage of fault detection (APFD) values for the baselines and the proposed techniques, aggregated on projects.

are assessed with the same score by MTK, are scheduled one after another. As a consequence, the other changed methods in a project version will be exercised only after a group of test cases with the same score is executed, typically leading to poorer results.

As an example, we presented the results of the re-ordered test suite obtained by the application of MTK prioritization in Table 3 (left) when considering the pair (V_1, V_2) of *Joda-time* on one of the 100 faulted variants available. As highlighted in the table, the first three tests have the highest score and are scheduled before test cases with lower values. Analyzing the coverage of these test cases, we noted that all three cover the same changed methods, as also their name suggests, that is, they belong to the same test class. None of these tests, however, discover faults, as they cover only methods which were not injected by any mutation in the variant. Thus, the execution of these test cases does not benefit the rate of fault detection of the prioritized suite. The fault F_1 in the variant is discovered by the fourth test case, which has a lower score, as it covers fewer changed methods. Subsequent test cases cover exactly the same changes and all discover the same fault F_1, so their execution, one after another, penalizes the APFD value of the permutation. The next valuable test case is in position 20, having a different score from the previous tests, and uncovers the fault F_2. Then, other test cases are assessed with the same score and scheduled to be executed after it until a new fault is discovered by other test cases with a different score. This phenomenon repeats until the end of the permutation.

This behavior has a particular impact on projects with a high number of test cases, such as *Joda-time* and *AssertJ*, and when a limited number of changes is present. In this case, indeed, it is more likely that many test cases execute the same changed methods, especially if the changes are mainly distributed in core classes used by different components. This causes a large number of tests to be assessed with the same score and to be executed in sequence in the prioritization, with the risk of depleting a significant amount of resources allocated for the test phase.

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, as the APFD value is significantly penalized by a delay in the discovery of faults. In fact, this behavior can be also seen in the results of the *total* and *diff* baselines.

The performances of MTK are generally worse than all the baselines, and the main reason is its redundancy issue. Our experimentation shows that the MTK approach will not give benefits in fault detection when compared with other prioritization techniques. It could thus not be successfully applied to RTP problems without further post-prioritization steps to correct its redundancy issue.

According to the results of our experiments, MTK-QS prioritization mitigates the redundancy issue of MTK. 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, we present in Table 3 (right) the prioritized test suite generated from MTK-QS in the same setting as the MTK example above. 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 case chosen by the MTK-QS prioritization belongs to the second, higher-scored equivalence class, while it would have been chosen as fourth by the MTK approach. 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 5, MTK-QS outperforms both MTK and the other baselines on all projects, with diff-QS being the baseline whose APFD results are closer. Although the quotient set approach, applied to the measure of code churn 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 than MTK. 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 different changed methods, and equivalence classes defined on it have smaller chances to exercise the same changes. This lead to the risk of scheduling test cases covering more changes later in the permutation. On the other hand, a QS prioritization on more refined code churn measures can lead to better diversification on coverage of changed methods, suggesting that the score assigned with TKs might be seen as a *fingerprint* of the set of changed units a test case covers. The diversification of coverage exhibited by MTK-QS is higher even compared with Genetic and ART-MaxMin 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 of changes in these techniques leads to worse APFD performances.

Figure 6 shows the results of the Wilcoxon–Mann–Withney *U*-test of the proposed techniques with respect to the baselines, after applying the Bonferroni correction. It is not typically possible to reject the null hypothesis for MTK prioritization when its APFD results are compared to other baselines, and in many cases, there is no significant difference in the produced permutations. MTK-QS, on the other hand, produced prioritized test suites which with higher fault detection rates for almost all projects, and the null hypothesis can be rejected with a high amount of confidence ($p \ll 0.05$, omitted for brevity). The only exception to the general trend is the *JOpt* project, where MTK-QS is not significantly better than the *diff* prioritization. In this project, however, all the considered techniques produced APFD values close to 1, which makes the produced permutations similar to each other in terms of fault detection. More generally, the observations suggest that, typically, MTK-QS prioritization tends to perform statistically better in terms of fault detection rate when the different baselines produce notably different permutations.

We also investigated the performances of the proposed techniques according to the extent of changes between the two versions in a pair. To this purpose, we partitioned the 53 experimental pairs into three subsets ranging on the number of changed methods between the previous

TABLE 3 Comparison on permutations produced by MTK (left) and MTK-QS (right) for an experiment on Joda-Time.

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
⋮	⋮	⋮	⋮

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
⋮	⋮	⋮	⋮

Note: Each row is related to a test case. The score column shows the score assigned to a test case by MTK and MTK-QS, respectively, while 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 left table shows sets of test cases with the same score.
Abbreviations: MTK, method-level tree kernel prioritization; MTK-QS, method-level tree kernel with quotient set.

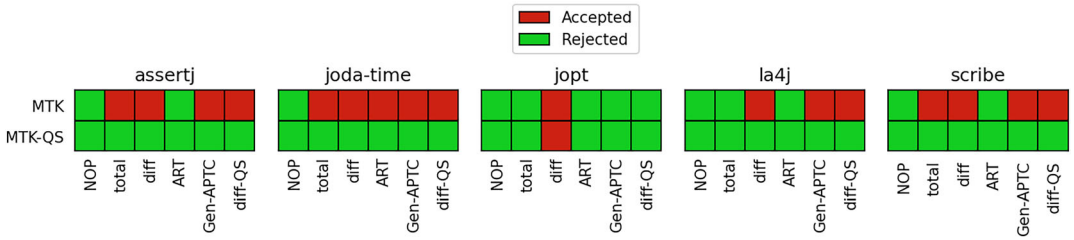

FIGURE 6 Results of Wilcoxon–Mann–Whitney *U*-test for statistical significance of the proposed techniques with respect to the baselines, aggregated on projects. The color of the cells means that the null hypothesis can be accepted (red) or rejected (green) between the technique on the row and the one on the column.

version V_i and the current version V_j. The first range includes pairs which incurred in *small* number of changes, that is, between 1 and 15 methods; the second interval comprehends a *medium* degree of changes, ranging from 16 to 40 changed methods; to the third interval belong pairs with a *large* number of changed methods, that is 41 and more. The upper bound of this range is 117, which is the maximum number of changed methods between all the considered experimental pairs. We then evaluated the APFD metric on the aggregation of pairs falling in these intervals. This analysis disregards project characteristics, focusing on the behavior of the proposed technique only in relation to the extent of changes.

Figure 7B shows the distribution of experimental pairs in the intervals highlighting the project they belong to. As can be seen, the pairs are nearly equally distributed between the ranges. Pairs from almost all projects fall into all the ranges, with the exception of the smallest projects *JOpt* and *Scribe*, as their limited size do not allow greater extent of changes. Larger projects, such as *Joda-Time* and *AssertJ*, present pairs with heterogeneous degrees of changes, from small modifications to large patches involving several changes between the versions. This suggests that this aggregation can help us to analyze the behavior of our techniques on projects with various characteristics when changes of different entities are realized.

Figure 7A presents the APFD values of the proposed techniques and the baselines aggregated by small, medium and large numbers of changed methods. MTK results show that the approach has worse performances with respect to the other baselines. The redundancy in coverage rate is again the cause of these low results. In fact, when there is a small number of changes, the risk that test cases are entitled with the same score is high and therefore the execution of uncovered methods is delayed. On the other hand, when the number of changes increases, so 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 nonfaulted methods could be executed before subsequent cases eventually show the other faults.

These observations are also justified by the results of MTK-QS. In all the considered scenarios, this latter technique outperforms the MTK approach and it produces the best results precisely on pairs with the highest number of changes, with smaller variability in the APFD measures. The performance of MTK-QS is indeed better than all other considered techniques, both aware and unaware of changes, and also for all extents

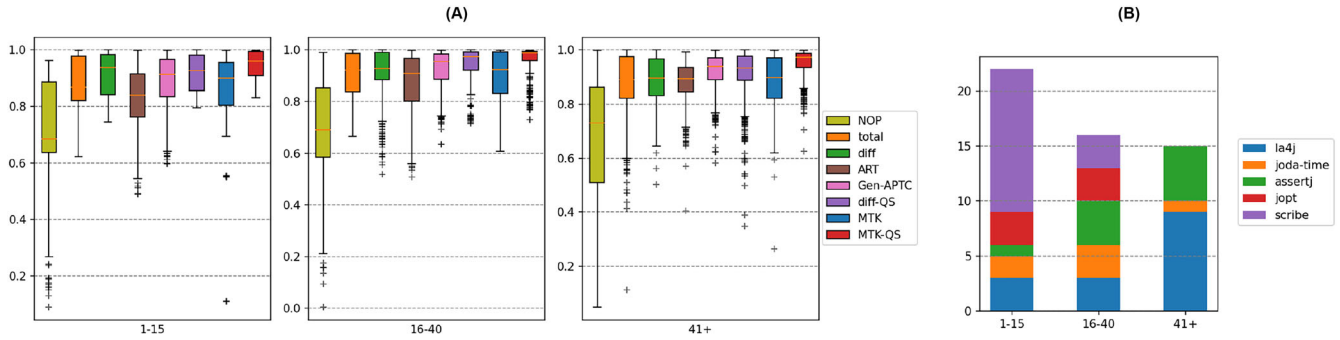

FIGURE 7 Results of analysis related to the number of changed methods between versions in an experimental pair. (A) average percentage of fault detection (APFD) values for subject projects aggregated in small (1–15), medium (16–40), and large (41+) numbers of changed methods. (B) Distribution experimental pairs in changed methods intervals divided by project.

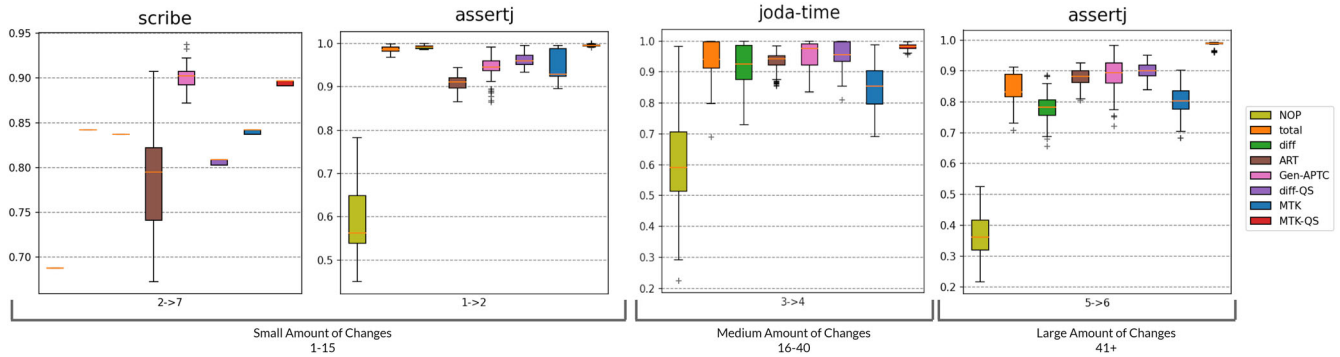

FIGURE 8 Average percentage of fault detection (APFD) values for specific experimental pairs divided by change interval (small, medium, and large).

of changes. It is also possible to note, again, that the improvement of MTK-QS with respect to MTK is higher than the application of quotient set to the diff technique. In particular, when considering small projects, the diff-QS performs slightly worse than diff. When the number of changed methods is low, several test cases tend to have low diff scores and therefore the score distribution is narrow. When this happens, the round-robin policy of quotient set causes a delay in the scheduling of test cases which actually cover a larger number of changed methods. The phenomenon is reduced when considering greater degrees of change, as the diff scores of test cases have better-diversified values. This suggests again that the combination of quotient set with MTK scores is more meaningful, producing permutations with high rates of fault detection, and also proves itself more stable than the other approaches.

To further drill down our observation on the techniques, we present in Figure 8 an in-depth analysis of four experimental pairs, with results aggregated on the 100 faulted variants produced. Each pair is denoted by the name of the project in which it belongs, and by the relative order of versions in our datasets which were used to construct the pair. We chose two pairs from the small changes interval and one pair for both medium and large intervals. As a note, in the small changes interval, the boxes are narrower, due to the fact that groups of injected faults among variants tend to be more homogeneous. This is the consequence of the small number of changes between the versions, which provide few methods in which the faults can be injected and, therefore, less variety in the variants. Only Genetic and ART produce wider values, due to their non-deterministic nature. As it is possible to see from these plots, diff-QS performs worse than the simple diff approach in both cases due to the small number of changes, while MTK-QS improves significantly the APFD results compared with MTK. When analyzing in detail the results on *scribe*, it is possible to see that the ART technique outperforms all others. This phenomenon could be ascribed to the specific considered variant, as the general trend of ART for the *scribe* project is different (see Figure 5).

Considering the experimental pair with medium and large changes, that is, *Joda-Time* and the *AssertJ* pairs, MTK is typically worse than all other techniques on all variants. Conversely, MTK-QS outperforms all other baselines and has higher results even on the variants in which it produces its worst APFD results. As the projects have large test suites, the MTK-QS scheduling with a high rate of fault detection can remarkably benefit the verification phase.

To analyze how many resources can be saved by employing the proposed techniques compared with the baselines, we evaluated the PTF and PTL faults metrics. The results obtained for these metrics are presented in Figure 9. For each technique and each project, two overlapping

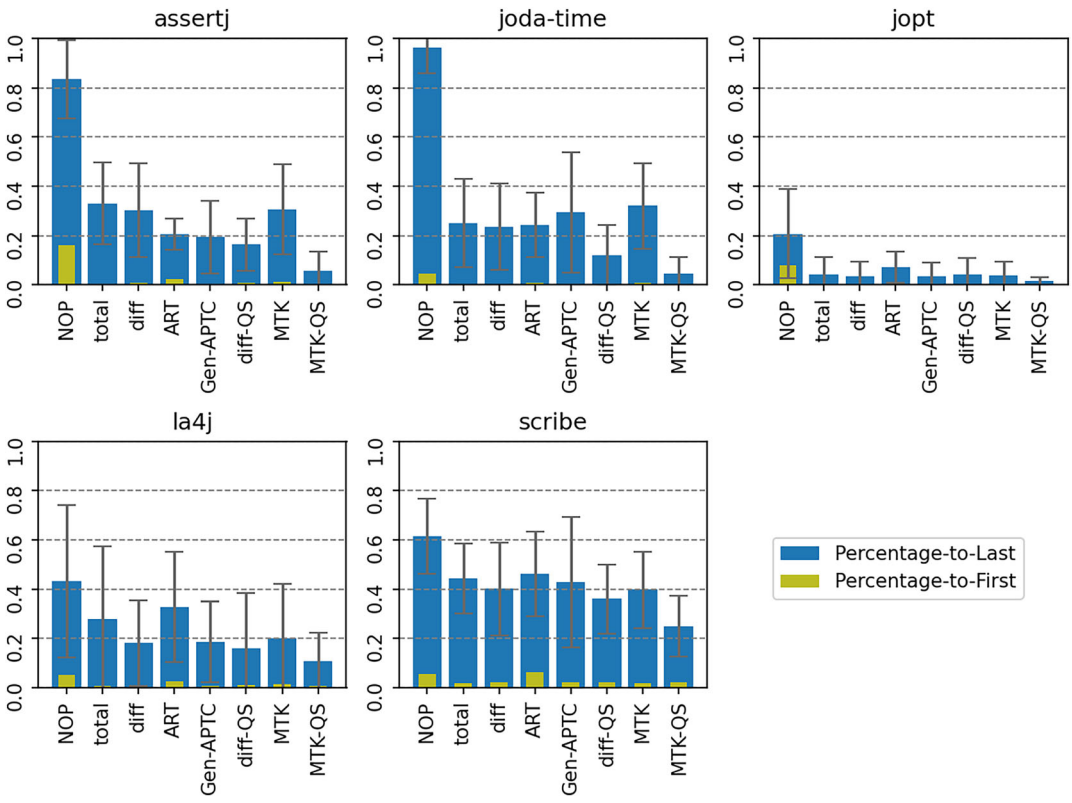

FIGURE 9 Bar plots showing percentage-to-first (PTF) and Percentage-to-Last (PTL) fault metrics for the baselines and the proposed techniques, aggregated on projects.

bars have been drawn, a green bar with the average PTF value and a blue bar with the corresponding PTL result. Each bar presents PTF and PTL values aggregated on all pairs and variants in a project for a specific technique. Furthermore, for the PTL metric, we also included the standard deviation error bar atop, as this measure showed more variations in the results.

As it is possible to see, all the considered techniques have typically a low result of PTF metric, with the exception of the untreated test suite (*NOP*). This is in line with the observation concerning the APFD stated above, that is, that the application of any prioritization technique can bring benefits to the testing activity.

Analyzing the results of MTK, it can be seen that the PTF is usually small, meaning that even the first test cases in the produced permutation can exhibit at least one fault. The PTL metric, on the contrary, is typically higher than the other baselines. This is again imputable to the poor distribution of coverage of changed methods, and thus, the application of the quotient set gives significantly lower values of PTL. MTK-QS, in fact, performs better among all the other techniques even in discovering all faults introduced in a pair of versions. Specifically focusing on larger projects, it can be seen that the fraction of the test suite which needs to be executed to discover all the faults are below the 10% on almost all variants.

6.2 | Analysis of execution time

To perform the benchmark on execution times, we adopted the JMH framework and collected the benchmark results for all techniques. Each benchmark involved 30 repetitions for each technique, with two independent instances of the JVM to reduce the chance of noise and inconsistencies in the evaluated results. We then excluded the times for ART and Genetic baselines as their execution time was typically 2 orders of magnitude longer than other techniques and therefore not applicable in practice. The evaluated times do not include the time for loading the coverage reports, as all the techniques rely on coverage information to be executed. We also excluded the computational time related to the detection of changed files between the pair of versions in an experiment, as this information is in practice evaluated by the VCS employed for the projects and thus it is available at no cost.

Figure 10 shows the results of the execution of benchmarks on the subject projects. As it can generally be seen, the amount of time for the execution of all strategies varies according to both the project size and the number of test cases. For smaller projects as *Scribe* and *JOpt*, the

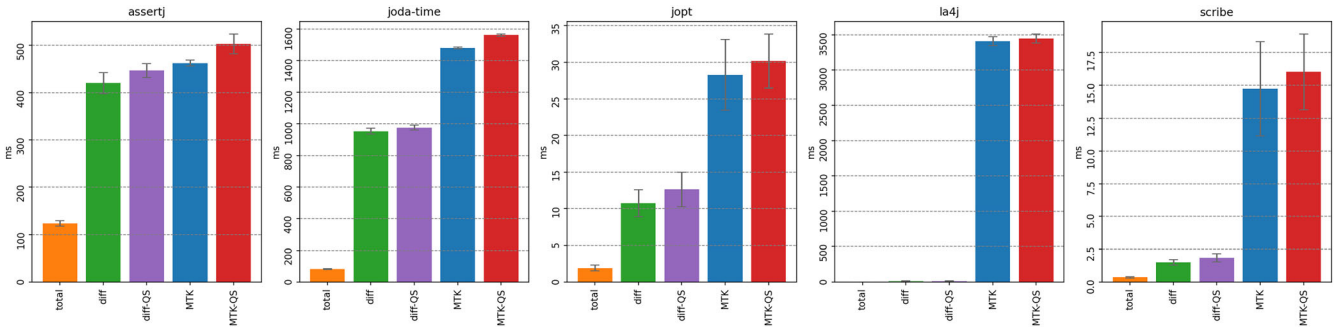

FIGURE 10 Comparison of the execution time of the proposed techniques and the baselines on subject projects, considering the transition to the first and last version available for all projects. The average execution time for 30 repetitions of the techniques is reported. Error bars show the standard deviation of the execution time distributions.

execution times are set in the range of few to few tens milliseconds, while for larger projects (*Joda-Time* and *AssertJ*) the execution time is significantly higher.

For three out of the five projects, that is, *AssertJ*, *JOpt*, and *Joda-Time*, the execution times of TK-based techniques are comparable, introducing only little relative overheads with respect to the baselines. This scenario suggests that the evaluation time of structural similarity of changed methods using TK functions has a higher correlation with the number of changes between the versions, rather than the size of the project. Considering the *Scribe* subject project, the execution time of MTK and MTK-QS is sensibly higher, with times around 1 order of magnitude longer. This difference is ascribable to the small size of the test suite of the project, which causes the time for the evaluation of similarity to dominate both the scoring and the sorting processes of the other baselines. However, the execution time for the proposed techniques is still lower than 25 μs, and it is 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 2 orders of magnitude higher than the baselines. The main cause of this is not attributable to the size of the project or to the number of test cases in its suite, but rather to the nature of the project itself. In fact, *La4J* is a linear algebra library for Java, and it is characterized by methods with long sequences of instruction at the same nesting level, usually for performance reasons. In particular, two of its methods, `decompose` and `hqr2`, are composed of 465 and 462 lines of code, respectively, with only few nested blocks and several statements which are executed in sequence. For this reason, these methods produce ASTs with a large branching factor, therefore having a large impact on the evaluation of similarity. For this reason, a TK-based prioritization technique might not be suitable for projects with these characteristics. However, according to the general cases and good practices of modern developmental methods, *La4J* is an atypical subject and similar scenarios should be quite rare in practice. In fact, best common practices in literature suggest to not exceed 20 lines of code in a method,[70] which *La4J* does not follow due to its nature. Notice that other methods up to 200 lines of code, which are present in other projects as *AssertJ* and *Joda-Time*, are processed in a few milliseconds.

It can also be seen that the application of QS prioritization using both MTK and diff scores introduces a negligible overhead, as it includes only the round-robin selection of tests in the equivalence classes. In fact, if the scored test suite fed to QS is already sorted in descending order of assigned scores, the approach just needs to sequentially scan the permutation and pick the test cases in turn.

To analyze the benefits of the application of MTK and MTK-QS techniques compared with the baselines, we performed an evaluation of the *effective execution time* metrics of the various strategies, as defined in Equation (9).

To this end, we first recorded the execution times of each technique for each experimental pair, using the *JMH* framework. To obtain a more robust measure, we also included the time to parse coverage reports in the benchmark. Then, we extracted the produced permutations for each variant and summed up the execution time of test cases in the order scheduled by each permutation, until all faults were exhibited at least once.

To retrieve the execution time of various test cases, we executed the testing phase of all projects and versions on the same hardware that was used to evaluate the execution time benchmarks. The execution time of test cases has been extracted from the *Surefire* test execution reports automatically produced after the completion of the testing phase. As the resolution of the Surefire reports was 1 μs, an execution time of 0 was assigned to some test cases and thus the evaluated effective execution times for all the techniques can be slightly lower than the real ones. Furthermore, the test suite execution times widely varied both between projects and versions in the same project, which could lead to hardly comparable results. To mitigate these problems, we applied a normalization process to the evaluated effective times. In particular, we normalized the effective execution time by the total test suite time, obtaining values in the range $[0,1]$.

To assess the statistical significance of our findings, we also performed the Wilcoxon–Mann–Whitney *U*-test on the effective execution times. This analysis focuses on how the effective execution time of the proposed MTK and MTK-QS techniques compared with the other baselines, that is, whether the effective execution times are significantly shorter or longer than the baselines. To this end, we performed two different Wilcoxon–Mann–Whitney tests, with the following *null hypotheses*:

$L_0^{t_1,t_2,p}$: 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.
$G_0^{t_1,t_2,p}$: 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.

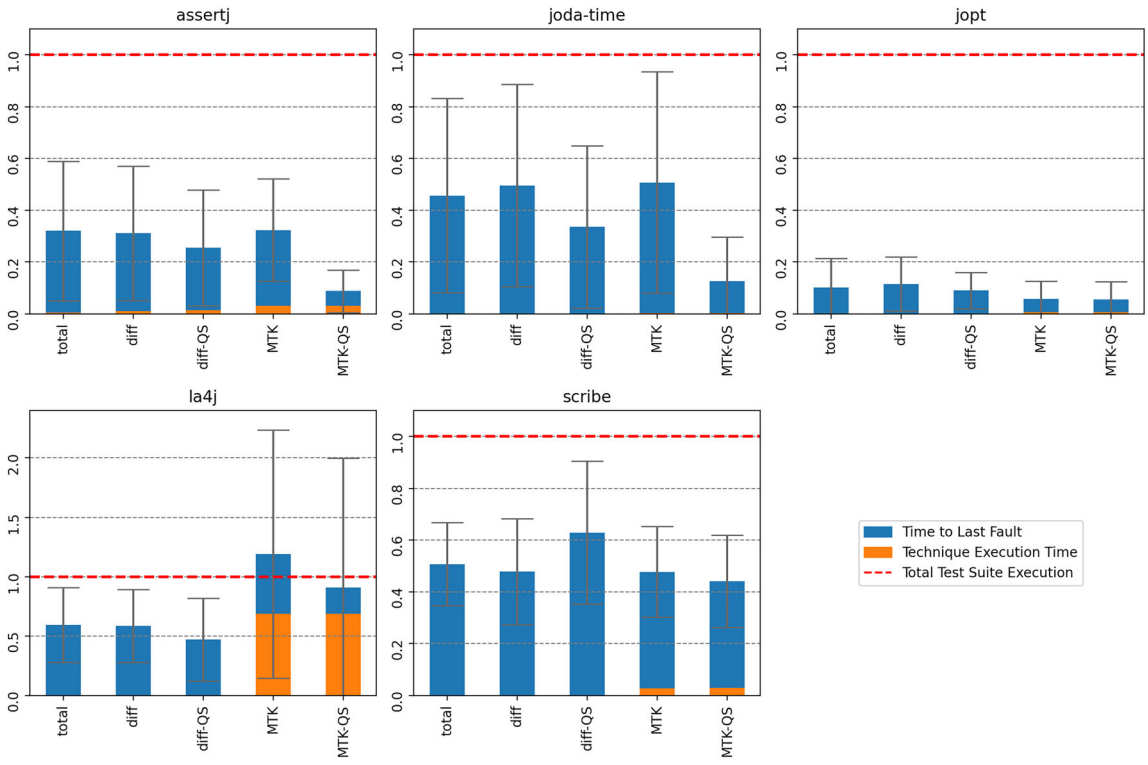

FIGURE 11 Averages of the effective time elapsed by techniques between projects. The plots include both the execution time of techniques and the total time consumed by the test cases in the permutation until all faults have been discovered. The values are normalized by the total execution time of the whole test suite, which is represented as the horizontal red line at a height equal to 1.

As we did in Section 6.1, we set the confidence value $\alpha = 0.05$ and applied the same *Bonferroni* correction to mitigate the *multiple comparisons problem* (α/K, where K is the number of hypotheses) to each statistical tests.

Figure 11 shows the effective time of the various techniques, aggregated by project. Bars are related to the *average* effective times between versions and variants in each project, and error bars show their standard deviation. We highlighted, in each bar, the time employed by the application of the technique (in orange) and by the time of the subsequence of test cases in the produced permutations which discovered all faults (in blue). We also discarded *ART-MaxMin* and *Genetic* prioritization, as their execution times alone in almost all cases exceeded the whole test suite execution time.

As it can be seen from Figure 11, 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. When the execution time of the technique is also considered, on average, the effective time of MTK is longer than the time to execute the whole test suite, while MTK-QS is slightly lower than the suite, but the performance is much higher than the other baselines. It is also possible to see that the standard deviation of the proposed techniques is very high. This is due to the high variance of the time required to MTK and MTK-QS to produce a permutation on the various versions of *La4J*. Indeed, when the large methods in the project have not been changed between the two versions, the actual evaluation of the PTK on them is not performed, and the execution time is significantly shorter. As in some pair of versions the larger methods are unchanged, in these cases, the techniques produce a lower effective time value than pairs in which these larger methods have been modified, thus increasing the variance of the results. For all the pairs in *La4J*, however, MTK and MTK-QS on *La4J* do not produce any benefit in the effective execution time with respect to the baselines.

FIGURE 12 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. A gray color indicates that the distributions do not differ in a statistically significant way.

For *Scribe* and *JOpt* projects, it is possible to see that both techniques produce results 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, shorter effective times, with permutations which provide more benefits to the testing activity.

On *AssertJ* and *Joda-Time* projects, the MTK and MTK-QS execution time is again negligible, assessing to the 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 in our dataset, but it can be seen that the techniques introduce small overheads to the testing phase. Thus, the effective time is dominated by the slice of the test case permutation which should be executed to uncover all faults. Concerning MTK, we can see that the effective execution time is longer than the baseline approaches, due to its redundancy issues, and performs poorly in comparison with these techniques. On the other hand, the effective execution time of MTK-QS is significantly shorter than all the other techniques on these two projects. In particular, the second best-performing technique, that is, diff-QS, produces effective execution times at least doubled in comparison with 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 our findings about effective execution times, we performed two Mann–Whitney–Wilcoxon *U*-tests, whose results are summarized in Figure 12. A green cell in the figure means that we can reject with high confidence the null hypothesis $\mathbf{L}_0^{t_1,t_2,p}$, that is, the technique t_1 (on the row) has significantly less effective times than the technique t_2 (on the column) on a particular project p. On the other hand, a red cell means that the null hypotheses $\mathbf{G}_0^{t_1,t_2,p}$ can be rejected, and the effective times of the technique on the row are significantly longer than those of the technique on the column for the project. A gray cell, instead, means that none of the two hypotheses can be confidently rejected and the effective times of the techniques are not significantly different.

For the *AssertJ* and *JOpt* projects, the MTK approach produces effective times comparable with the total and diff baselines, while being slower than the diff-QS approach to exhibit all faults in the current version. Indeed, the MTK-QS approach produces significantly shorter effective execution times with a very high level of confidence for both projects, and it can be used to save more time in the testing phase than the other approaches.

On the *JOpt* projects, both MTK and MTK-QS techniques perform significantly better than the considered baselines. In fact, we can reject the null hypothesis $\mathbf{L}_0^{t_1,t_2,p}$ for all the other techniques. However, all the techniques have small values of effective execution times and thus the time spared by MTK and MTK-QS is minimal. A similar scenario happens for *Scribe*. The MTK approach is better than the total and the diff-QS baselines while producing similar effective times compared with the diff. MTK-QS is indeed significantly better than the others. As happened with *JOpt*, also in this case all the techniques produce similar effective times, so the gaining in terms of time is not remarkable.

Concerning *La4J* project, the proposed techniques perform significantly worse than all the baselines. In fact, it is possible to reject $\mathbf{G}_0^{t_1,t_2,p}$ for all the comparisons with the baseline techniques, meaning that the effective execution time is always longer than the other techniques. This result is in line with the analysis of the average effective time and it is due to the heavy execution time of the technique in presence of extremely large methods, which are present in the project. Thus, the usage of MTK and MTK-QS is not recommended for *La4J*.

In general, the performance of MTK in terms of effective times was good on the smaller projects (*JOpt* and *Scribe*), while it produced results which were similar to the baselines on the largest ones. On the other hand, MTK-QS outperforms all techniques in almost all scenarios, bringing notable benefits for the testing phase. MTK-QS performed particularly well on the projects with a great size and a high 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. This observation comes with a *caveat*, as both the proposed techniques were definitely unsuitable for the *La4J* project. From our analysis, it can be noted that exceedingly long methods with hundreds of lines of code are the main weakness of the techniques. 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.

7 | THREATS TO VALIDITY

In this section, we state threats that could have affected the results and the generalizability of the experiments we performed. To decrease the chance of misinterpretations, we followed widely used guidelines presented in literature.[71]

7.1 | Threats to internal validity

Internal threats refer to the presence of confounding variables limiting the confidence of the association between the treatment conditions and the establishment of the results. The first threat is related to the implementation of TK prioritization. To reduce the possibility of defects in our implementation, we relied on *GumTreeDiff* library to generate AST and to the *KeLP* implementation of TK functions. These libraries are well-maintained and consolidated in literature, ensuring no unexpected behavior will rise. Furthermore, coverage reports are obtained via *OpenClover*, which is widely used in practice to obtain coverage reports. Our addition to the reporting tool reuses the *OpenClover* inner reporting module, extending it toadd per-test information to each method, which the tool already evaluates in its coverage collection process. In our implementation, to reconstruct coverage information we just read the XML report produced by the tool. The model produced by the reading module is used with all the baselines, to share the same coverage information between these techniques.

Moreover, the *total* prioritization, which we re-implemented from scratch, was designed closely following the original definition provided by the authors. The *diff* prioritization was entirely implemented, but we relied on the *git diff* utility to obtain information on which methods were changed between the version, to limit the chance of mismatch in the performed evaluation of changes. Additionally, the code implementing the baselines was revised by the authors and tested in a controlled environment.

To reduce the discrepancies between the QS-based prioritizations, which use different scoring functions, we designed the QS module using a high level of abstraction. The implementation of the technique receives directly the test suite whose test cases were labeled with the respective ranking evaluated by MTK or Diff, respectively. Thus, the same code is executed when evaluating MTK-QS and diff-QS.

Concerning the internal threats to collecting the execution times, we employed JMH, a robust and well-known framework to assess performance testing, to reduce the chance of noise in the evaluation of times. It is however possible that uncontrollable external factors could rise fluctuation in the registered data, due to JVM optimizations and to the operating system routines. To mitigate these effects and ensure fairer results, we adopted several best practices[72] in writing the benchmark code. For the effective execution time analysis, we used the times provided by the Surefire test execution reports. To guarantee the consistency between technique execution time and the time elapsed by test cases, we executed all the test suites on the same machine we used for the technique execution times benchmarks. Furthermore, due to the resolution of execution times in Surefire reports being 1 millisecond, a time of 0 was assigned to some test cases which employed less time. Thus, it is possible that the evaluated effective execution times were lower than the real ones. However, this phenomenon seldom occurred in our dataset, and to mitigate its impact, we normalized these values by the execution time of the whole test suite.

An additional threat to the validity of our conclusions and statistical inferences is posed by the *multiple comparisons problem*. Indeed, when performing multiple statistical tests for a family of hypotheses, the likelihood of obtaining false-positive results increases because the more comparisons made, the higher the probability of observing statistically significant results by chance alone. This can lead to the erroneous identification of significant effects or relationships. To mitigate this threat, as done in a number of other software engineering studies,[68,69] we adopted the Bonferroni correction. The Bonferroni correction adjusts the significance threshold for each individual test, ensuring that the overall family-wise error rate remains controlled.

7.2 | Threats to external validity

These threats can reduce the generalization of our findings in the results. For the empirical evaluation of this study, we employed four open-source Java projects already used as benchmark projects for regression testing prioritization in literature. We selected these projects to be as heterogeneous as possible in their size, intended as the number of statements, ranging from a small number to projects with larger sizes. 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 were not available in the benchmark projects. This approach has been employed in many works on the prioritization topic[14,15] and can be appropriate for research when the environment is controlled.[62] Nevertheless, mutation faults might not be representative of real faults in some contexts.[73] To mitigate this, we injected the faults only in methods which were already changed between the software versions considered in the experiment. In this case, the injected faults can be viewed as mistakes by developers when rewriting the methods. Furthermore, to diversify as much as possible the types of faults, we activated all mutation operators provided by the *Major* tool.

In order to reduce these threats and enhance the generalization of our findings, we are currently working on further experimental scenarios employing real-world faults and a larger number of software projects.

8 | CONCLUSIONS

RTP is an important topic in Software Engineering, and it is widely used in practice in software evolution scenarios. In this study, we proposed two novel prioritization approaches which leverage a refined evaluation of the source code changes, assuming that changes can introduce faults.

The first approach is MTK, a RTP strategy which evaluates the structural similarity of the source code between two software versions, using TK functions on the AST representation of the source code to quantify the amount of changes. This information is merged with code coverage metrics to assign a score to each test case based on the structural changes of the methods it covers, and eventually the test suite is re-ordered accordingly.

The second proposed technique is MTK-QS prioritization, an evolution of the MTK approach designed to achieve a better rate of coverage on changed methods. MTK-QS evaluates a quotient set of the test suite, obtained by partitioning the set of test cases in equivalence classes according to their scores evaluated by MTK, and prioritizes the suite picking tests in turn from these classes. The quotient set can be applied to any designed test case scoring function but achieved higher performances in our experiments when used atop MTK scores.

We empirically evaluated the performances of the two techniques using common fault detection metrics on a dataset composed of 25 software versions across five real-world Java projects with assorted characteristics, in which we injected faults through software mutation. Among these versions, we defined 53 experimental evolutionary scenarios in which our approaches could be executed and compared the results with five widely used prioritization approaches. To assess the feasibility of the proposed techniques, we also evaluated their execution time on each subject project. The results of our experiments showed that the application of MTK on the subject projects can identify the fault-exhibiting test cases according to the structural similarity between the changes, but produces permutations with low fault detection rates, due to the redundancy in the scheduling of tests. In fact, MTK tends to schedule test cases covering the same changed methods consecutively, with the risk of depleting testing resources stressing the same code units. For this reason, the use of MTK alone will not produce benefits over the baseline techniques and should be applied with other correction steps, such as the quotient set approach.

On the other hand, MTK-QS showed better performances both in APFD rate, PTF and PTL metrics, producing also stabler results. The analysis of the execution times for the proposed techniques also showed that the approaches are generally applicable in real contexts, introducing small overheads compared to the baseline techniques in four out of five subject projects. The only exception is *La4J*, a linear algebra library for Java, where the execution times of our approaches are significantly longer than all other baselines. The main cause is the peculiar nature of this project, which features a development process which is substantially different from modern and widely adopted industrial developmental practices, as it is characterized by long methods. For projects with this peculiarity, the evaluation of TK functions takes a longer time since longer methods are more likely to produce wider ASTs.

In all other cases, the analysis showed that our techniques scale well even when applied to large software applications. Our results suggest that, in practice, MTK alone is comparable with or worse than the baselines in most cases, and it cannot be effectively used due to its redundancy. On the other hand, MTK-QS can be satisfyingly applied in real-world contexts, aiding developers to enhance the benefits of a software verification phase even in contexts with limited resources available.

Our next research lines involve an extension of our experimentation on datasets of projects exhibiting *real faults*, rather than subjects with artificial faults. This can be useful to assess the effectiveness of the proposed approaches in an in vivo environment and to evaluate their feasibility. We are actually retrieving subjects from popular datasets with these characteristics,[74,75] furnishing them with additional information needed to apply our techniques and simultaneously extending our set of employed projects to construct a meaningful dataset on which experiment our approaches.

In future works, we are also planning to further refine the evaluation of similarity, in order to investigate the impact of the *semantics* of changes on prioritization, along with structural changes analyzed in this study. More in detail, we are working on more evolved tree kernel functions which take into account also the similarity of the specific programming language constructs connected to AST fragments. In the tree kernel function family, there are specific functions capable to include this information in the similarity evaluation, for example, *Smoothed TKs*. They provide a higher number of parameters which can be used to give different weights to tree fragments with the same structure but distinct semantical meanings, and all these parameters can be tunable to highlight specific features for the task at hand. As we plan to also consider the semantics of changes, we will also design an approach to analyze the occurrences of method renaming, to obtain more refined matches.

In parallel, while we have assessed the benefits of the tools we employed for RTP with this study, we want to refine the level of our change analysis. To this purpose, we aim to improve our code analysis tool by leveraging a finer-grained statement-level granularity, rather than working at method level. We envision that a more accurate analysis employing finer code units can lead to further improvement in the effectiveness of our techniques.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Replication Package Repository at https://zenodo.org/record/8256619, reference number 10.5281/zenodo.8256619.

ORCID

Francesco Altiero https://orcid.org/0000-0001-7090-4249
Anna Corazza https://orcid.org/0000-0002-9156-5079
Sergio Di Martino https://orcid.org/0000-0002-1019-9004
Adriano Peron https://orcid.org/0000-0002-7111-3171
Luigi Libero Lucio Starace https://orcid.org/0000-0001-7945-9014

REFERENCES

1. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Softw Test Verif Reliab*. 2012;22(2):67-120. https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.430
2. Coviello C, Romano S, Scanniello G, Antoniol G. Gasser: genetic algorithm for test suite reduction. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM); 2020:1-6.
3. Chen L. Continuous delivery: huge benefits, but challenges too. *IEEE Softw*. 2015;32(2):50-54.
4. Memon A, Gao Z, Nguyen B, et al. Taming Google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP); 2017:233-242.
5. Elbaum S, Rothermel G, Penix J. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2014:235-245.
6. Bagherzadeh M, Kahani N, Briand L. Reinforcement learning for test case prioritization. *IEEE Trans Softw Eng*. 2021;48(8):2836-2856.
7. Sun X, Li B, Zhang S, Tao C, Chen X, Wen W. Using lattice of class and method dependence for change impact analysis of object oriented programs. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11. Association for Computing Machinery; 2011; New York, NY, USA: 1439-1444. https://doi.org/10.1145/1982185.1982495
8. Li B, Sun X, Leung H. Combining concept lattice with call graph for impact analysis. *Adv Eng Softw*. 2012;53:1-13.
9. Wang S, Liu T, Nam J, Tan L. Deep semantic feature learning for software defect prediction. *IEEE Trans Softw Eng*. 2020;46(12):1267-1293.
10. Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16. Association for Computing Machinery; 2016; New York, NY, USA:297-308. https://doi.org/10.1145/2884781.2884804
11. Hoang T, Khanh Dam H, Kamei Y, Lo D, Ubayashi N. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR); 2019:34-45.
12. Corazza A, Di Martino S, Maggio V, Scanniello G. A tree kernel based approach for clone detection. In: 2010 IEEE International Conference on Software Maintenance; 2010:1-5.
13. Ishikawa T, Liu Y-L, Shepard DL, Shin K. Machine learning for tree structures in fake site detection. In: Proceedings of the 15th International Conference on Availability, Reliability and Security; 2020:1-10.
14. Lu Y, Lou Y, Cheng S, et al. How does regression test prioritization perform in real-world software evolution? In: Proceedings of the 38th International Conference on Software Engineering; 2016:535-546.
15. Luo Q, Moran K, Zhang L, Poshyvanyk D. How do static and dynamic test case prioritization techniques perform on modern software systems? An extensive study on github projects. *IEEE Trans Softw Eng*. 2018;45(11):1054-1080.
16. Engström E, Runeson P. A qualitative survey of regression testing practices. In: International Conference on Product Focused Software Process Improvement International Conference on Product Focused Software Process Improvement; 2010:3-16.
17. Chittimalli PK, Harrold MJ. Recomputing coverage information to assist regression testing. *IEEE Trans Softw Eng*. 2009;35(4):452-469.
18. Harrold MJ. Testing evolving software. *J Syst Softw*. 1999;47(2):173-181. https://www.sciencedirect.com/science/article/pii/S0164121299000370
19. Kazmi R, Jawawi DNA, Mohamad R, Ghani I. Effective regression test case selection: a systematic literature review. *ACM Comput Surv (CSUR)*. 2017;50(2):1-32.
20. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Trans Softw Eng*. 2001;27(10):929-948.
21. Khatibsyarbini M, Isa MA, Jawawi DayangNA, Tumeng R. Test case prioritization approaches in regression testing: a systematic literature review. *Inform Softw Technol*. 2018;93:74-93.
22. Hao D, Zhang L, Zhang L, Rothermel G, Mei H. A unified test case prioritization approach. *ACM Trans Softw Eng Methodol (TOSEM)*. 2014;24(2):1-31.
23. Rothermel G, Untch RH, Chu C, Harrold MJ. Test case prioritization: an empirical study. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99); 1999:179-188.
24. Hao D, Zhang L, Zang L, Wang Y, Wu X, Xie T. To be optimal or not in test-case prioritization. *IEEE Trans Softw Eng*. 2015;42(5):490-505.
25. Jones JA, Harrold MJ. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans Softw Eng*. 2003;29(3):195-209.
26. Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: a family of empirical studies. *IEEE Trans Softw Eng*. 2002;28(2):159-182.
27. Huang R, Zhang Q, Towey D, Sun W, Chen J. Regression test case prioritization by code combinations coverage. *J Syst Softw*. 2020;169:110712.
28. Beszédes A, Gergely T, Schrettner L, Jász J, Langó L, Gyimóthy T. Code coverage-based regression test selection and prioritization in webkit. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM); 2012:46-55.
29. Altiero F, Corazza A, Di Martino S, Peron A, Starace LLL. Inspecting code churns to prioritize test cases. *IFIP International Conference on Testing Software and Systems*: Springer International Publishing; 2020:272-285.
30. Korel B, Tahat LH, Harman M. Test prioritization using system models. In: 21st IEEE International Conference on Software Maintenance (ICSM'05); 2005:559-568.
31. Korel B, Koutsogiannakis G. Experimental comparison of code-based and model-based test prioritization. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops; 2009:77-84.
32. Kim J-M, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th International Conference on Software Engineering (ICSE); 2002:119-129.
33. Park H, Ryu H, Baik J. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In: 2008 Second International Conference on Secure System Integration and Reliability Improvement; 2008:39-46.

34. Lin C-T, Chen C-D, Tsai C-S, Kapfhammer GM. History-based test case prioritization with software version awareness. In: 2013 18th International Conference on Engineering of Complex Computer Systems; 2013:171-172.
35. Noor TB, Hemmati H. A similarity-based approach for test case prioritization using historical failure data. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE); 2015:58-68.
36. Vescan A, Şerban C. Towards a new test case prioritization approach based on fuzzy clustering analysis. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME); 2020:786-788.
37. Zhang C, Chen Z, Zhao Z, Yan S, Zhang J, Xu B. An improved regression test selection technique by clustering execution profiles. In: 2010 10th International Conference on Quality Software; 2010:171-179.
38. Wang R, Qu B, Lu Y. Empirical study of the effects of different profiles on regression test case reduction. *IET Softw*. 2015;9(2):29-38.
39. Palma F, Abdou T, Bener A, Maidens J, Liu S. An improvement to test case failure prediction in the context of test case prioritization. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'18. Association for Computing Machinery; 2018:80-89. https://doi.org/10.1145/3273934.3273944
40. Jiang B, Zhang Z, Chan WK, Tse TH. Adaptive random test case prioritization. In: 2009 IEEE/ACM International Conference on Automated Software Engineering; 2009:233-244.
41. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Trans Softw Eng*. 2007;33(4):225-237.
42. Huang Y-C, Peng K-L, Huang C-Y. A history-based cost-cognizant test case prioritization technique in regression testing. *J Syst Softw*. 2012;85(3):626-637.
43. Gao D, Guo X, Zhao L. Test case prioritization for regression testing based on ant colony optimization. In: 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS); 2015:275-279.
44. Nayak S, Kumar C, Tripathi S, Mohanty N, Baral V. Regression test optimization and prioritization using honey bee optimization algorithm with fuzzy rule base. *Soft Comput*. 2021;25:9925-9942.
45. Xing Y, Wang X, Shen Q. Test case prioritization based on artificial fish school algorithm. *Comput Commun*. 2021;180:295-302.
46. Kaur A, Bhatt D. Hybrid particle swarm optimization for regression testing. *Int J Comput Sci Eng*. 2011;3(5):1815-1824.
47. Busjaeger B, Xie T. Learning for test prioritization: an industrial case study. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2016:975-980.
48. Bertolino A, Guerriero A, Miranda B, Pietrantuono R, Russo S. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Association for Computing Machinery; 2020; New York, NY, USA:1-12.
49. Peng Q, Shi A, Zhang L. Empirically revisiting and enhancing ir-based test-case prioritization. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2020:324-336.
50. Yusuff SR, Bajeh AO, Aro TO, Adewole K. Improving the accuracy of static source code based software change impact analysis through hybrid techniques: A review. *Int J Softw Eng Comput Syst*. 2021;7(1):57-66.
51. Alenezi M, Magel K. Empirical evaluation of a new coupling metric: Combining structural and semantic coupling. *Int J Comput Appl*. 2014;36(1):34-44.
52. Neamtiu I, Foster JS, Hicks M. Understanding source code evolution using abstract syntax tree matching. In: Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR), MSR '05. Association for Computing Machinery; 2005; New York, NY, USA:1-5. https://doi.org/10.1145/1083142.1083143
53. Fan Y, Xia X, Lo D, Hassan AE, Wang Y, Li S. A differential testing approach for evaluating abstract syntax tree mapping algorithms. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) IEEE; 2021:1174-1185.
54. Hoang T, Kang HJ, Lo D, Lawall J. Cc2vec: Distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE), ICSE '20. Association for Computing Machinery; 2020; New York, NY, USA:518-529. https://doi.org/10.1145/3377811.3380361
55. Haussler D. Convolution kernels on discrete structures. *Technical Report*. UCS-CRL-99-10, University of California at Santa Cruz; 1999.
56. Collins M, Duffy N. Convolution kernels for natural language. *Adv Neural Inform Process Syst*. 2002;14:625-632. https://doi.org/10.7551/mitpress/1120.003.0085
57. Moschitti A. Making tree kernels practical for natural language learning. In: 11th Conference of the European Chapter of the Association for Computational Linguistics; 2006:113-120.
58. Moschitti A. Efficient convolution kernels for dependency and constituent syntactic trees. Lecture Notes in Computer Science (including subseries LectureNotes in Artificial Intelligence and Lecture Notes in Bioinformatics); 2006:318-329.
59. Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools*. 2nd ed.: Pearson Education, Inc; 2006. https://dl.acm.org/doi/10.5555/1177220
60. Filice S, Castellucci G, Martino GDS, Moschitti A, Croce D, Basili R. Kelp: A kernel-based learning platform. *J Machine Learn Res*. 2018;18(191):1-5. http://jmlr.org/papers/v18/16-087.html
61. Altiero F, Corazza A, Martino SD, Peron A, Starace LLL. *Replication Package for "Regression Test Prioritization Leveraging Source Code Similarity With Tree Kernels"*: Zenodo; 2023. https://doi.org/10.5281/zenodo.8256619
62. Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing?. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2014:654-665. https://doi.org/10.1145/2635868.2635929
63. Just R. The major mutation framework: Efficient and scalable mutation analysis for java. In: 2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings; 2014:433-436.
64. Mei H, Hao D, Zhang L, Zhang L, Zhou J, Rothermel G. A static approach to prioritizing junit test cases. *IEEE Trans Softw Eng*. 2012;38(6):1258-1275.
65. Zhang L, Hao D, Zhang L, Rothermel G, Mei H. Bridging the gap between the total and additional test-case prioritization strategies. In: 2013 35th International Conference on Software Engineering (ICSE); 2013:192-201.
66. Falleri J-R, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: ACM/IEEE International Conference on Automated Software Engineering (ASE); 2014:313-324. http://doi.acm.org/10.1145/2642937.2642982
67. Lou Y, Chen J, Zhang L, Hao D. A survey on regression test-case prioritization. In: Memon AM, ed. *Advances in Computers*, Vol. 113: Elsevier; 2019:1-46. https://www.sciencedirect.com/science/article/pii/S0065245818300615

68. Sarro F, Petrozziello A, Harman M. Multi-objective software effort estimation. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), ICSE '16. Association for Computing MachineryICSE'16; 2016; New York, NY, USA:619-630.
69. Mittas N, Angelis L. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Trans Softw Eng*. 2012; 39(4):537-551.
70. Martin RC. *Clean Code: A handbook of Agile Software Craftsmanship*. 1st ed. USA: Prentice Hall PTR; 2008.
71. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B. *Experimentation in Software Engineering*: Springer; 2012.
72. Costa D, Bezemer C-P, Leitner P, Andrzejak A. What's wrong with my benchmark results? Studying bad practices in JMH benchmarks. *IEEE Trans Softw Eng*. 2021;47(7):1452-1467.
73. Paterson D, Campos J, Abreu R, Kapfhammer GM, Fraser G, McMinn P. An empirical study on the use of defect prediction for test case prioritization. 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST); 2019:346-357.
74. Altiero F, Corazza A, Di Martino S, Peron A, Starace LLL. Recover: A curated dataset for regression testing research. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR). Association for Computing Machinery; 2022:196-200.
75. Mattis T, Rein P, Dürsch F, Hirschfeld R. Rtptorrent: An open-source dataset for evaluating regression test prioritization. In: Proceedings of the 17th International Conference on Mining Software Repositories (MSR), MSR '20. Association for Computing Machinery; 2020; New York, NY, USA:385-396. https://doi.org/10.1145/3379597.3387458