Change-Aware Regression Test Prioritization using Genetic Algorithms

Francesco Altiero, Giovanni Colella, Anna Corazza, Sergio Di Martino, Adriano Peron and Luigi L. L. Starace
Università degli Studi di Napoli Federico II, DIETI, Naples, Italy
Email: {francesco.altiero, giovan.colella, anna.corazza, sergio.dimartino, adrperon, luigiliberolucio.starace}@unina.it

***Abstract*—Regression testing is a practice aimed at providing confidence that, within software maintenance, the changes in the code base have introduced no faults in previously validated functionalities. With the software industry shifting towards iterative and incremental development with shorter release cycles, the straightforward approach of re-executing the entire test suite on each new version of the software is often unfeasible due to time and resource constraints. In such scenarios, Test Case Prioritization (TCP) strategies aim at providing an effective ordering of the test suite, so that the tests that are more likely to expose faults are executed earlier and fault detection is maximised even when test execution needs to be abruptly terminated due to external constraints.**

In this work, we propose *Genetic-Diff*, a TCP strategy based on a genetic algorithm featuring a specifically-designed crossover operator and a novel objective function that combines code coverage metrics with an analysis of changes in the code base. We empirically evaluate the proposed algorithm on several releases of three heterogeneous real-world, open source Java projects, in which we artificially injected faults, and compare the results with other state-of-the-art TCP techniques using fault-detection rate metrics. Findings show that the proposed technique performs generally better than the baselines, especially when there is a limited amount of code changes, which is a common scenario in modern development practices.

***Index Terms*—Regression Testing, Test Case Prioritization, Genetic Algorithms**

I. Introduction

Within the software evolution process, changes in the source code can introduce faults not only in the new or modified features, but also in previously validated functionalities not affected by changes. In the literature, this phenomenon is referred to as *Software Regression*. *Regression Testing* is a set of activities aimed at providing confidence that (I) the changed parts of the software behave as intended, and (II) the unchanged parts have not been adversely affected by the modifications [1]. Although extensively applied in industry, regression testing is challenging from both a process management and a resource management perspective. Indeed, the most straightforward regression testing approach, which consists in re-running the entire test suite of the software under development after each evolution step, in the so-called *retest-all* fashion [2], is often prohibitively time consuming. This is especially true with novel software development methodologies, such as *Continuous Integration/Continuous Delivery* (CI/CD), where new releases can be very frequent [3]. As a consequence, many research efforts have been dedicated to propose different approaches for regression testing, with the general goal of reducing the required testing efforts, while at the same time keeping a high confidence on the quality of the software [4]. Among them, Test Case Prioritization (TCP) is one of the most investigated and widely-used family of solutions. Indeed, TCP techniques aim at identifying an '*ideal*' ordering of test cases that maximizes some desirable properties, such as fault detection rate. Thanks to the adoption of such a meaningful ordering, the benefits of the testing activity would be maximised, even when test execution is abruptly interrupted before completion, due to resource constraints [5]. Finding such an optimal ordering, however, is not a trivial task, as it is impossible to know in advance, before execution, which tests will fail and expose faults. Hence, researchers in the field have proposed various heuristic techniques to estimate the fault-revealing capability of a test and approximate an optimal ordering.

In the TCP literature, many strategies have been proposed to define the test case ordering. Many prior works have investigated the use of test coverage as a proxy for the fault-revealing capability of a test, under the assumption that tests covering more code are also more likely to expose faults, and thus must receive higher priority. Other works have proposed to exploit *code churn* information, i.e. changes introduced in the current version of the code base.

In this paper, we propose *Genetic-Diff*, a TCP solution aimed at combining two different sources of information, i.e.: test coverage and code churn metrics, to define a test case ordering. Our proposal searches for an ordering of the test cases in which the *changed* parts of code are covered as early as possible, under the assumption that changed code is more likely to expose faults, while at the same time favouring test cases that stress large parts of the source code. Specifically, *Genetic-Diff* leverages a genetic algorithm with a specifically-designed crossover operator and a novel objective function combining fine-grained code coverage information at statement level with an analysis of code changes (i.e.: *code churn*) in the current version.

To understand the effectiveness of the proposed strategy, we performed an empirical evaluation based on several releases of three real-world software systems featuring extensive test suites. More in detail, for each of these releases, we generated 100 variants by automatically injecting five different faults in the code base. Then, on each variant, we prioritized the test suite with the proposed technique and several of state-of-the-art techniques previously used in the literature as a benchmark, comparing them based on the achieved fault detection rate.

Results show that *Genetic-Diff* performs better than the benchmark techniques in a statistically significant way in most cases, especially when there is a limited amount of changes between the two considered versions, which is the most common scenario in modern CI/CD environments.

The paper is organized as follows. In Section II we provide preliminary definitions and survey related works on TCP, while in Section III we describe in detail the *Genetic-Diff* approach we propose. Section IV describes the experimental evaluation we conducted to assess the effectiveness of the proposed solution and reports on the results of our experiments. Finally, in Section V, we give concluding remarks and future research directions.

II. Definitions and Related Works

Regression testing aims at validating modified software and ensuring that no new errors are introduced into previously tested code. It is a consolidated industrial practice in the maintenance of evolving software [6], and can account for as much as one-half of the cost of software maintenance [7], [8]. Since a naive *retest-all* approach consisting in the re-execution of the entire test suite after each update of the code base might be too expensive or unfeasible in presence of resource constraints, a number of different approaches have been studied to ease regression testing efforts. Test Case Prioritization is one of such approaches [9], and aims at identifying an *ideal* ordering of test cases that maximizes desirable properties, such as the fault detection rate. Unlike other approaches such as test suite minimization and test case selection, test case prioritization preserves the integrity of the original test suite and can help maximize the benefits of the testing activity, even in case of abrupt termination due to external constraints [5].

Ideally, the desirable property to be maximised is early fault detection. However, one generally does not know which tests reveal faults until the test suite is executed. Hence, test case prioritization techniques are typically based on heuristic approaches leveraging surrogate properties to guide the search for a satisfactory ordering [5], [9].

A good amount of existing research focused on using test coverage information to guide the prioritization [4], under the assumption that early maximization of source code coverage can lead to early maximization of a higher chance of revealing faults. Code coverage can be considered at different granularity levels, including statement coverage [10], branch coverage [11], modified conditions/decision coverage [12], and function/method coverage [1]. A well-known coverage-based strategy, largely used as a baseline in many prioritization works, is the *total* strategy [13], that prioritizes test cases by sorting them in decreasing order of *total* number of source code units they cover.

Some researchers have proposed model-based prioritization approaches [14]. These works leverage state-based abstractions (models) of the system under test, and simulate the execution of the test suite on the models to gather coverage information at state or transition level [14]. Model-based prioritization approaches have been found to be promising in improving early fault detection and reducing the prioritization overhead [15]. However, these techniques require accurate system models to be effective.

Other works are based on the idea that historical project information, such as past test case failures, can be used to infer which test cases have a higher probability of failing in new releases. In [16], for example, the authors propose a technique that assigns a priority rank to a given test case based on its failures in past execution history. [17] further refined these intuitions by proposing a prioritization strategy that assigns more importance to recent information. The empirical evaluations conducted in these studies showed that considering also historical project information can improve the test prioritization results. However, these techniques may not be well adapted to continuously changing testing environments with frequent changes in code and test suites [5].

A class of works in the literature has also investigated a strategy aimed at prioritizing test cases so that the most '*diverse*' tests according to some suitably-defined similarity measure are preferred. Many of these similarity-based approaches are based on clustering analysis [18], [19], i.e., partition test cases into clusters based on the similarity of their execution profiles. A different approach is defined in [20], which introduces Adaptive Random Test Prioritization. This technique starts by selecting the first test to execute in a random way, and then proceeds by selecting the most diverse test among those that are not already selected, until no test case remains. In their original work, the distance between two test cases is the Jaccard distance computed on the two respective sets of covered structural code units.

Machine-learning based prioritization approaches have also been recently investigated. For example, [21] presents an approach that integrates multiple existing techniques, including coverage, history, and similarity-based ones, via a systematic machine learning framework to rank task cases. More recently, [22] conducted an empirical study comparing ten different machine learning prioritization techniques, including supervised and reinforcement-learning base approaches, in a continuous integration setting. These techniques were proved to be possibly effective, but require a non-trivial training process involving a large volume of data that in some cases might not be available [23].

Some studies [24], [25] have also investigated the use of source code changes (i.e.: *code churn*) information in prioritization. In [24], the authors considered procedure-level test coverage information and devised a strategy focused at covering the procedures that were affected by changes in the new version of the software. More recently, in [25], the authors proposed several coverage-based prioritization strategies that take into account also code change information. Preliminary experiments reported in that work confirm that considering code change information might help improve the effectiveness of prioritization. However, the coverage of changed parts was optimized using a simple greedy algorithm, and hence significant improvements might be possible with a more-refined optimization strategy.

Regardless of the adopted surrogate properties, finding the permutation of the test suite that optimizes it is a difficult optimization problem. Many works have employed meta-heuristic algorithms to search for an optimal solution in the space of permutations of the test-suite. Commonly-used meta-heuristic search algorithms include *Particle Swarm* [26], *Artificial Fish School* [27], and *Genetic Algorithms* [28], which are among the most widely used optimization technique in TCP. The work presented in [28] was among the first to explore these approaches, by comparing in their empirical study 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 [29], which also leveraged history-based information. More recently, [30] proposed a Hypervolume-based Genetic Algorithm, namely HGA, to solve the Test Case Prioritization problem when using multiple test coverage criteria.

To the best of our knowledge, all the existing prioritization techniques either do not consider code change information at all, or do so in a limited way. In this work, we present a novel prioritization strategy that combines code coverage information and code churn to estimate the fault revealing capability of a permutation of a test suite, and uses a genetic algorithm we developed, namely *Genetic-Diff*, to effectively search for ideal permutations.

III. The Proposed Prioritization Approach

In this section, we present *Genetic-Diff*, a novel TCP solution based on a Genetic Algorithm leveraging code coverage and code churn information to search for a permutation of the test suite which maximizes the coverage rate for changed statements. We first describe the intuition behind the proposal, then provide a brief recall on Genetic Algorithms, and finally detail the proposed solution.

A. Rationale

Our idea to face the Regression Test Prioritization (RTP) problem is to define an ordering by combining test coverage information and code changes between two versions of a software. Indeed, our idea is that test coverage information should be integrated with details on the nature of the coverage, under the assumption that test cases covering more *churned* lines of code should have higher priority, as these changes might introduce regression faults. For this reason, our goal is to produce a test case permutation that covers changed statements as quickly as possible.

As for the generation of the permutation, heuristic and meta-heuristic techniques are widely used in the field, as they can efficiently explore a wide solution search-space. In particular, we opted to use *Genetic Algorithms*, which are a very common approach in the RTP literature. The key motivation is that genetic algorithms present a high level of flexibility, allowing us to extensively tailor their application to the problem at the hand. In the next section we briefly recall the key points of genetic algorithms.

B. Genetic Algorithms

Genetic algorithms are a class of optimization techniques that draw inspiration from the evolutionary theory to explore the search-space of a problem and to find a possibly optimal solution. Starting from an initial (often random) population of individuals, i.e. a set of encoded solutions for the specific problem, they simulate the evolution of the population, where only the fittest give rise to offspring. To this aim, it is necessary to define a suitable *fitness function* to evaluate the goodness of a solution/individual. The goal of the algorithm is to find a solution for which this function is maximized.

To evolve the population to a next generation, fittest individuals are *mutated*, or bred through a *crossover operation*. The *mutation* operation consists into a random alteration of a solution, to widen the search space, while the *crossover* operator starts with two solutions, i.e. the *parents*, and mixes their encoding to produce one or more new solutions. A *selector* operator then is used to discard solutions with lower fitness value and to produce a population for the next generation. The process is repeated until one or more ending condition is met (e.g., a certain number of generations has been reached).

For TCP problems, usually a population is a set of permutations of the test suite. The *encoding* used to represent a permutation is a *sequence encoding*, which is an array whose size is equal to the number of tests in the suite, and the i-th cell of the array stores the identifier of the test having position i in the permutation.

C. Design and Implementation

In our approach, to combine test coverage with code-churn information, we designed our own fitness function and a crossover operator. In particular, as fitness function, we defined the *Average Percentage Transition Coverage on Difference* ($APTC_{\text{diff}}$), an extension of the classical and widely-used *Average Percentage Transition Coverage* (*APTC*). While classical APTC considers the coverage rate of all code units, the custom version we designed, $APTC_{\text{diff}}$, given a test suite, measures the rate at which it covers the *changed* code units.

The code churn, i.e. the lines of code changed between two software versions, can be evaluated by using any text difference tool, such as the *Diff* utility. In particular, a line is considered as changed from the previous version if it has been edited or removed. Let us note that we cannot consider lines added in the newer version of the source code, as no coverage information is available to those lines.

Thus, given a test suite $\mathcal{TS}$, its $APTC_{\text{diff}}$ fitness function is evaluated as:

$$APTC_{\text{diff}}(\mathcal{TS}) = 1 - \frac{\sum_{i=1}^{m} TC(i)}{n \cdot m} + \frac{1}{2n} \qquad (1)$$

In Equation 1, m refers to the total number of changed lines, n to the total number of tests in the suite and $TC(i)$ is the index of the first test case in the suite which covers the i-th changed line.

To create the new offspring, genetic operators (i.e., crossover and mutation) are applied within a certain probability to a current population. In particular, starting from the population $\mathcal{P}$ in a generation of the genetic algorithm, we use the standard *fitness-proportionate selector* operator [31] to select the individuals in the population to be used for breeding, according to a probability proportional to its fitness value.

As *crossover* operator to breed two permutations, we designed *ChurnPriorityCrossover*. This operator leverages coverage and churn information to determine where the genes of parents should be placed in offspring. Starting from the first parent P_1 and for its first test case, the operator checks if it was already inserted in the child permutation. If not, it is checked whether the test covers any unit in the code-churn, in which case it is inserted in the first available position of the child from the beginning, or else is put in the first available position on the tail. After that a test in P_1 has been placed, the same operation is applied to a test in the second parent P_2, in turn. Figure 1 shows an example of the *ChurnPriorityCrossover*, considering a test suite with five test cases, labeled from 1 to 5. Test cases 1, 3 and 4 cover some code units in the code churn in the version pair. Given the two parents P_1 and P_2, the first iteration of the algorithm evaluates the test case 1 from P_1 and test case 3 from P_2. As test case 1 covers the code churn, it is placed in the first position of the child. Test case 3 in P_2 is placed in the next position, as it also covers elements of the code churn. In the second iteration, test cases 2 and 1 from P_1 and P_2, respectively, are evaluated. Test case 2 does not cover any element in the code churn, thus it is placed in the last position of the child, while as test case 1 from P_2 has already been inserted, it is skipped. In the third iteration, test cases 3 and 2 are considered but are both skipped as they are already in the child. When evaluating test cases 4 and 5, the first is put after test 3, as it covers the code churn, while test case 5, not covering the code churn, takes place behind the test 2. In the last iteration, test cases 5 and 4 are processed and are again both skipped as already present in the child. Eventually, the produced child is the test order 1, 3, 4, 5 and 2. It can be noted that the tests covering code churn have higher priority with respect to those not covering any code churn element.

Algorithm 1 presents the pseudo-code of *ChurnPriorityCrossover*. Given two parents P_1 and P_2, it is possible to obtain two children by this operator, simply switching the order of parents.

To mutate individuals, we employed a *Swap Mutator* operator. Given a permutation, each test case has a probability to be swapped with another test case in the permutation. The probability of a swap has to be small, as with higher probabilities the operator degenerates in a simple random shuffling of the permutation. We assigned to each individual a 10% chance of being selected for mutation.

To select the test suite permutations which will survive and be included in the next generation, we applied a *Linear Rank Selector*. The operator assigns a rank to the test suite in the population according to their fitness value and then assigns a probability based on its ranking. The surviving solution are then selected accordingly to these probabilities, which are higher for solutions with a better fitness score.

Algorithm 1: The *ChurnPriorityCrossover* algorithm.

```
Data: $P_1, P_2$: parents, $N$: test suite size
Result: $child$: offspring of $P_1$ and $P_2$
$child = []$;
$i \leftarrow 1$;
$frontPos \leftarrow 1$;
$backPos \leftarrow N$;
while $i \leq N$ do
    if $P_1[i] \notin child$ then
        if coversChurn($P_1[i]$) then
            $child[frontPos] \leftarrow P_1[i]$;
            $frontPos \leftarrow frontPos + 1$;
        else
            $child[backPos] \leftarrow P_1[i]$;
            $backPos \leftarrow backPos - 1$;
    if $P_2[i] \notin child$ then
        if coversChurn($P_2[i]$) then
            $child[frontPos] \leftarrow P_2[i]$;
            $frontPos \leftarrow frontPos + 1$;
        else
            $child[backPos] \leftarrow P_2[i]$;
            $backPos \leftarrow backPos - 1$;
    $i \leftarrow i + 1$;
return $child$;
```

IV. Empirical Evaluation

The goal of our empirical evaluation is to assess whether *Genetic-Diff*, the technique we propose, performs better than baseline prioritization techniques. To this end, we selected three open source software projects and, as done in many other studies, automatically injected faults in their codebase. Then, we applied *Genetic-Diff* and the baseline techniques and measured how fast the automatically-injected faults are detected by the resulting prioritized test suites.

In this section, we start by describing in detail our experimental protocol in terms of employed data, procedure, and metrics. Then, we present the results of the empirical evaluation. Lastly, we discuss some threats that might limit the validity and generalizability of our findings.

A. Data and Procedure

To perform the empirical evaluation of the technique we propose, we selected three open-source Java projects which were also used in previous works in TCP [32], [33]. We selected these projects for two reasons: (1) they have been employed in a number of other studies on TCP; (2) they vary both in size (i.e.: lines of code (LoCs)) and in the number of tests cases in their suite, improving the generalizability of the results. Details on the employed projects are shown in Table I.

For our empirical evaluation, we followed an approach similar to the one used in [32], performing prioritization between several versions of a project with its last version. That is, for each version V_i of a project, we considered the changes between V_i and V_n as an incremental step in the evolution, where n is the number of the most recent

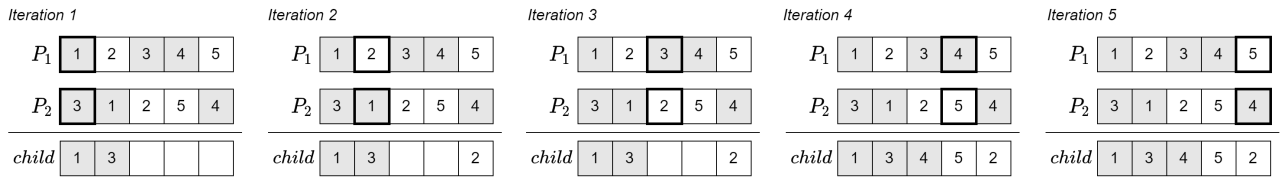


Fig. 1: Example of the *ChurnPriorityCrossover* operator for a test suite of five test cases. Shading in cells means that a test covers churned code. Bold-edged cells are the elements in the parents which are evaluated in each iteration.

Project	Avg. LoCs	Avg. Tests	Shared Tests	Description
AssertJ	60k	2419	1401	Assertion framework for Java.
JOpt	7k	427	319	Argument parsing library.
Metrics	11k	294	201	Tool to compute software metrics.

TABLE I: Projects used in our empirical evaluation.

version of the project. We believe that this methodology can be considered as representative of modern *Agile* software development practices, in which iterations can span different amount of time, ranging from a few days to one or two months, and the entity of the changes varies accordingly. Comparing different versions with the latest one allows to simulate these different scenarios. More in detail, for each of the considered projects, we selected four contiguous software versions, which we denote with ids ranging from 1 to 4, with 1 being the least recent version and 4 being the most recent one.

Since the projects do not contain any real fault, we injected faults via *code mutation*: this is a technique widely employed to assess the quality of a test suite [34]. The mutation process consists in changing parts of the source code of a software, e.g., substituting one arithmetic operator with another, changing the value of literals or altering logic operators in conditional blocks. Each change in the code is referred to as *mutant*. When a test is executed on the mutated software, it can either discover the mutation (i.e., the test fails) or be not affected by the mutation and passes normally.

To prepare the setting for our experiments, for each project we mutated the source code of its last version (i.e.: version 4). We denote the versions with injected faults as $4F$. The mutations were injected only in those methods which have been changed between versions and are common to all the versions. We used the *Major*[1] mutation framework to create the mutants, as it mutates directly the Java source code rather than its Bytecode. We then executed the test phase to select only those mutants which were discovered by at least one test. We then defined 100 mutated variants, each of them with 5 uniformly and randomly selected mutants which were considered as the faults for that variant.

We implemented the *Genetic-Diff* algorithm using the Java programming language and the well-known Jenetics library[2]. *Jenetics* allows to readily implement and design different types of meta-heuristics, including genetic algorithms. It provides great flexibility and ease to use, allowing also a parallel execution of algorithms in an user-transparent fashion. We executed the Genetic-Diff algorithm on all pairs of versions (V_i, V_n) for all projects and all variants. The population size was set to 100 and the number of iterations to 200, as these parameters are widely used in empirical evaluation of genetic algorithms for test prioritization [28], [32]. We included also experiments the version pairs $4 \rightarrow 4F$, only for evaluation purposes, as there is no change in the source code but the faults which were injected.

To evaluate the quality of the prioritization, we used the *APFD* metric [1]. *APFD* is a widely used metric in prioritization studies to evaluate the rate of fault detection of a permuted test suite, and is defined as follows:

$$APFD(TS) = 1 - \frac{\sum_{i=1}^{m} TF(i)}{n \cdot m} + \frac{1}{2n} \qquad (2)$$

where m is the number of faults discovered by the test suite, n is the number of test cases and $TF(i)$ is the index of the first test case in the prioritized suite which discovers the i-th fault. Note that the definition of *APFD* is conceptually quite similar to the definition of $APTC_{Diff}$ given in Equation 1. The latter, however, is a proxy measure for test failures and is defined only in terms of covered changed statements, measuring how quickly the changed parts the code base are covered. *APFD*, on the other hand, is defined in terms of the real faults that are discovered, and actually measures how quickly the real faults are discovered.

We compared the *APFD* results of Genetic-Diff with those of three prioritization techniques which were used as baselines:

- *Total*: this technique gives higher priority to tests which cover a greater number of code units, such as statements or methods.
- *Adaptive Random Test Prioritization* (*ART*): chooses the next test in the permutation according to different coverage granularity and to different function to evaluate the distance between test cases previously prioritized.
- *Genetic-APTC*: a genetic algorithm which uses *Average Percentage of Covered Statements* as target function, a random crossover function to generate offspring and a random swap-mutator as mutation function. For our study, we set the population to 100 and the number of iteration to 200. The algorithm was repeated once for each variant, giving a total of 100 repetitions.

[1]https://mutation-testing.org/, visited on 03/04/2022.
[2]https://jenetics.io/, visited on 02/04/2022.

For the *ART* and *Genetic APTC* prioritization we used the source code available in [32] and only implemented a minimal wrapper to integrate the baselines in our pipeline. As for the *Total* technique, we implemented it from scratch. For techniques which use coverage information, we considered the coverage report related to the earlier version in the version pair, as in real-world scenarios coverage data is available for the previous version only.

B. Results

In Table II, we report, for each of the considered software projects and version pairs, the average APFD value achieved by each of the considered techniques on the 100 generated variants. These figures highlight that *Genetic-Diff* performs better, on average, than the baselines in all the considered settings, with improvements w.r.t. the best-performing baseline ranging from approximately 2% to 10%, and averaging at approximately 5%.

These APFD results are also visualized using box plots in Figure 2, which highlights that *Genetic-Diff* performs better than the baselines in terms of median as well, and generally exhibits a smaller variance in APFD values among the generated variants. Moreover, from Figure 2, it is also easy to point out a general trend in the performance of *Genetic-Diff*, that improve when versions in the considered version pair are temporally closer. In our opinion, this is explained by the fact that coverage information of more recent versions provides more up to date information on which statements are covered by which test case. Hence, the fitness function and the crossover operator rely on more precise information which led to better results in terms of $APFD$.

In addition to the above described quantitative comparison on APFD, we also perform tests to assess whether the results are statistically significant. More in detail, for each subject application and version pair, and for each of the baseline techniques, we performed paired Mann-Whithney-Wilcoxon tests to verify the null hypothesis that the *APFD* achieved by our technique is not greater than that achieved by the baseline in the considered setting. More formally, for each software project s, version pair v and baseline technique b, we test the null hypothesis:

$\mathbf{H}_0^{s,v,b}$: The APFD achieved by *Genetic-Diff* on the version pair v of software project s is not greater than that achieved by b in the same setting.

The results of these statistical tests are presented in the tile plot in Figure 3 in which, for each software project s and version pair v, and for each baseline technique b, the corresponding cell contains the *p-value* resulting from the statistical test on $H_0^{s,v,b}$. Moreover, cells with *p-value* < 0.05, i.e., for which it is possible to reject the corresponding null hypothesis with high confidence, accepting the alternative hypothesis that *Genetic-Diff* performs better than the baseline b, are colored in green, whereas the remaining cells are colored in red.

The statistical tests confirm that, in almost all the settings, the null hypotheses can be rejected with very high confidence (*p-value* $\ll 0.05$), thus accepting the alternative hypotheses that *Genetic-Diff* performs better than the baselines in a statistically significant way. The only exceptions arise when comparing our proposal against the ART baseline on *AssertJ*, version pair $2 \rightarrow 4F$, and when comparing against *Total* on JOpt, version pair $1 \rightarrow 4F$. In these cases, the statistical tests are inconclusive.

C. Threats to validity

In this subsection, we discuss some threats that could have affected the results of the experiments and their generalizability, according to the guidelines proposed in [35].

1) Threats to internal validity: These threats are concerned with the presence of confounding variables which might limit the confidence with which a cause-and-effect association between treatment condition and results can be established. To ensure a fair comparison between all the techniques, we made sure to allow each of them the same amount of time for termination, running them on the same server with the same load conditions. To ensure a fair comparison between the *Genetic-Diff* and the genetic algorithm we used as a baseline (*Genetic-APTC*), we ran both of them with the same meta-parameters (e.g.: population size, number of generations, etc.). Moreover, to perform this study, we re-implemented three TCP techniques presented in prior works, to use them as baselines. It is possible that there might be some differences between the original authors' implementations and our own. However, to mitigate this risk, we performed this task closely following the technical details presented in the original publications. When possible, we used the original source code provided by the authors and only implemented a minimal wrapper to integrate the baseline technique within our experimental pipeline. Additionally, two of the authors of this paper met for a detailed code review on the implemented baselines.

2) Threats to external validity: These threats limit the generalizability of the experimental results. In this study, we considered three open-source Java projects, which have been used as a benchmark in previous regression testing research. The set of considered software projects was selected to be as heterogeneous as possible, including small, medium, and large-sized projects. Nevertheless, the limited amount of considered projects might not be representative of all Java software. Moreover, since real faults were not available for the considered projects, we used artificially injected faults, as done in many other works on prioritization (e.g.: [32], [33]). Using artificially injected faults has been shown to be a suitable solution for regression testing research in controlled environments [34], but these faults might not be representative of real-world faults. The above limitations represent a threat to the possibility of generalizing our findings to every software and in real-world contexts. Further experiments involving more software, possibly with real-world faults, should be put in place to mitigate this threat.

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

TABLE II: Average APFD achieved by the considered techniques.

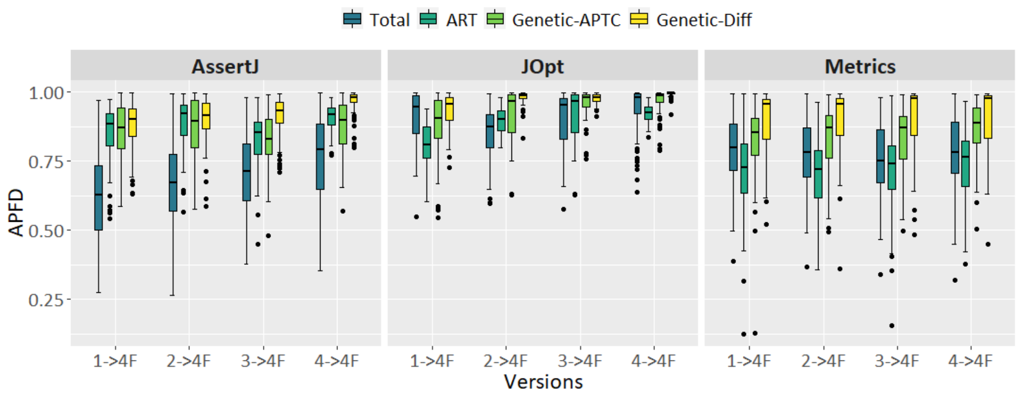
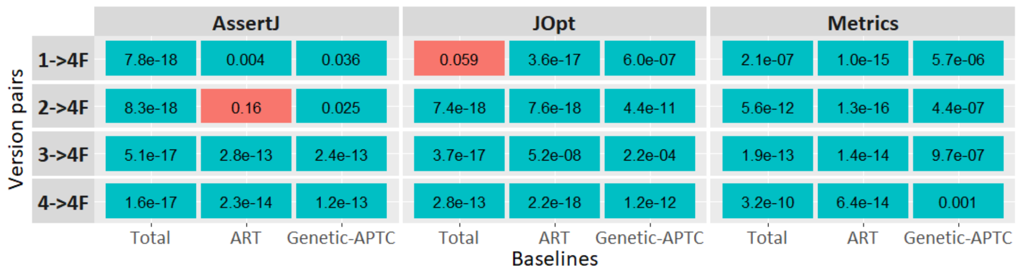


Fig. 2: Boxplots representing the APFD values achieved by the considered prioritization techniques.



	AssertJ Total	AssertJ ART	AssertJ Genetic-APTC	JOpt Total	JOpt ART	JOpt Genetic-APTC	Metrics Total	Metrics ART	Metrics Genetic-APTC
1->4F	7.8e-18	0.004	0.036	0.059	3.6e-17	6.0e-07	2.1e-07	1.0e-15	5.7e-06
2->4F	8.3e-18	0.16	0.025	7.4e-18	7.6e-18	4.4e-11	5.6e-12	1.3e-16	4.4e-07
3->4F	5.1e-17	2.8e-13	2.4e-13	3.7e-17	5.2e-08	2.2e-04	1.9e-13	1.4e-14	9.7e-07
4->4F	1.6e-17	2.3e-14	1.2e-13	2.8e-13	2.2e-18	1.2e-12	3.2e-10	6.4e-14	0.001

Fig. 3: Results of Mann-Withney-Wilcoxon tests with the statistical significance of comparison between *Genetic-Diff* and the baseline techniques.

V. Conclusions

Test Case Prioritization (TCP) is a widely-used industrial practice that can provide many benefits during the regression testing phase. Indeed, especially when release cycles are more frequent, employing suitable prioritization strategies to determine an ideal ordering of the test cases can lead to discovering faults earlier, with noticeable savings in both time and allocated computing resources.

In this work, we present *Genetic-Diff*, a novel test case prioritization approach based on an evolutionary strategy that leverages both code coverage and code churn information, aiming at prioritizing tests that cover parts of the software that underwent relevant changes. We assessed the effectiveness of the proposed approach by comparing it against state-of-the-art baselines on several releases of real-world software, in which we automatically injected faults. The proposed technique performed significantly better than the baselines in most of the scenarios.

In future works, we plan to investigate the effectiveness of the proposed technique on an extended dataset of real-world software systems with real faults, collected using the software repository mining tool we developed in [36]. Starting from these data, we plan to investigate real-world patterns of changes that are more critical. Subsequently, leveraging more advanced and customizable code churn evaluation techniques such as Tree Kernels [25], [37], we aim at developing a more refined approach capable of discriminating between less critical changes (e.g.: renaming of a variable or other trivial refactoring operations) and more critical ones (e.g.: structural changes that have an impact on the control flow). Moreover, we also plan to apply our prioritization technique in the different domain of E2E testing [38], where each test generally takes longer to execute w.r.t. unit tests, and prioritization becomes increasingly important.

References

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

[2] C. Coviello, S. Romano, G. Scanniello, and G. Antoniol, "Gasser: Genetic algorithm for test suite reduction," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–6.

[3] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[4] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[5] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, 2021.

[6] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *International Conference on Product Focused Software Process Improvement*. Springer, 2010, pp. 3–16.

[7] P. K. Chittimalli and M. J. Harrold, "Recomputing coverage information to assist regression testing," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 452–469, 2009.

[8] M. J. Harrold, "Testing evolving software," *Journal of Systems and Software*, vol. 47, no. 2, pp. 173–181, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121299000370

[9] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.430

[10] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 179–188.

[11] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2015.

[12] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[13] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–31, 2014.

[14] B. Korel, L. H. Tahat, and M. Harman, "Test prioritization using system models," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 559–568.

[15] B. Korel and G. Koutsogiannakis, "Experimental comparison of code-based and model-based test prioritization," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 77–84.

[16] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 119–129.

[17] C.-T. Lin, C.-D. Chen, C.-S. Tsai, and G. M. Kapfhammer, "History-based test case prioritization with software version awareness," in *2013 18th International Conference on Engineering of Complex Computer Systems*. IEEE, 2013, pp. 171–172.

[18] A. Vescan and C. Şerban, "Towards a new test case prioritization approach based on fuzzy clustering analysis," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 786–788.

[19] R. Wang, B. Qu, and Y. Lu, "Empirical study of the effects of different profiles on regression test case reduction," *IET Software*, vol. 9, no. 2, pp. 29–38, 2015.

[20] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 233–244.

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

[22] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "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*, 2020, pp. 1–12.

[23] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 324–336.

[24] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in webkit," in *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 2012, pp. 46–55.

[25] F. Altiero, A. Corazza, S. Di Martino, A. Peron, and L. L. L. Starace, "Inspecting code churns to prioritize test cases," in *IFIP International Conference on Testing Software and Systems*. Springer, 2020, pp. 272–285.

[26] M. Tyagi and S. Malhotra, "Test case prioritization using multi objective particle swarm optimizer," *2014 International Conference on Signal Propagation and Computer Technology, ICSPCT 2014*, pp. 390–395, 2014.

[27] Y. Xing, X. Wang, and Q. Shen, "Test case prioritization based on artificial fish school algorithm," *Computer Communications*, vol. 180, pp. 295–302, 12 2021. [Online]. Available: https://dl.acm.org/doi/abs/10.1016/j.comcom.2021.09.014

[28] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[29] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *Journal of Systems and Software*, vol. 85, no. 3, pp. 626–637, 2012.

[30] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "A test case prioritization genetic algorithm guided by the hypervolume indicator," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 674–696, 2018.

[31] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[32] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 535–546.

[33] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk, "How do static and dynamic test case prioritization techniques perform on modern software systems? an extensive study on github projects," *IEEE Transactions on Software Engineering*, vol. 45, no. 11, pp. 1054–1080, 2018.

[34] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, pp. 654–665, 11 2014. [Online]. Available: http://dx.doi.org/10.1145/2635868.2635929

[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[36] F. Altiero, A. Corazza, S. Di Martino, A. Peron, and L. L. L. Starace, "Recover: A curated dataset for regression testing research," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, May 2022.

[37] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "A tree kernel based approach for clone detection," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–5.

[38] S. Di Martino, A. R. Fasolino, L. L. L. Starace, and P. Tramontana, "Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1754, 2021.