Journal of Software Engineering Research and Development, 2023, 11:4, doi: 10.5753/jserd.2023.2142

An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration

Jackson Antonio do Prado Lima [Federal University of Paraná | jackson.lima@ufpr.br]
Silvia Regina Vergilio [Federal University of Paraná | silvia@inf.ufpr.br]

Abstract
Continuous Integration (CI) environments is a practice adopted by most organizations that allows frequent integration of software changes, making software evolution more rapid and cost-effective. Such environments require dynamic Test Case Prioritization (TCP) approaches that adapt better to the test budgets and frequent addition/removal of test cases. In this sense, Ranking-to-Learn approaches have been proposed and are more suitable for CI constraints. By observing past prioritizations and guided by reward functions, they learn the best prioritization for a given commit. In order to contribute for improvements and direct future research, this work evaluates how far the solutions produced by these approaches are from optimal solutions produced by a deterministic approach (ground truth). To this end, we consider two learning-based approaches i) RETECS, which is based on Reinforcement Learning; and ii) COLEMAN, an approach based on Multi-Armed Bandit. The evaluation was conducted with twelve systems, three test budgets, two reward functions, and six measures concerning fault detection effectiveness, early fault detection, test time reduction in the CI cycles, prioritization time, and accuracy. Our findings have some implications for the approaches application and reward function choice. The approaches are applicable in real scenarios and produce solutions very close to the optimal ones, respectively, in 92% and 75% of the cases. Both approaches have some limitations to learn with few historical test data (a small number of CI Cycles) and deal with a large test case set, in which many failures are distributed over many test cases.

Keywords: Test Case Prioritization, Continuous Integration environments, Ranking-to-Learn

1 Introduction

Continuous Integration (CI) is a common practice adopted by many organizations to make software evolution more cost-effective and reliable (Fowler, 2006). In a CI scenario, the software is changed, built, and tested many times in a short period. This is usually costly because a test suite often includes thousands of test cases and requires several hours or even days to execute (Haghighatkhah et al., 2018). In such a scenario, the application of regression testing techniques is fundamental.

Regression testing techniques are classified into three main categories (Yoo and Harman, 2012): minimization, selection, and prioritization. Techniques based on Test Case Minimization (TCM) usually remove redundant test cases, minimizing the test set according to some criterion. Test Case Selection (TCS) selects a subset of test cases, the most important ones to test the software. Test Case Prioritization (TCP) attempts to re-order a test suite to identify an "ideal" order of test cases that maximizes specific goals, such as early fault detection. TCP techniques are very popular in the industry and have some advantages because they do not discard any test case. Moreover, the test cases that have a higher probability of detecting a fault are executed first. This allows interrupting the test activity early and reducing costs and time between each CI cycle.

Existing TCP techniques use different kind of information to prioritize the test cases: historical failure data, test coverage, requirements, and system models (Yoo and Harman, 2012). However, most techniques need adaptations to be applied in CI environments. This is because in CI there are limited resources and constraints that can make the application of a technique infeasible, due to the time spent for the prioritization or code analysis. The application of TCP techniques that require extensive code analysis or coverage, such as search-based ones, is not always possible due to the test budget for a build. To deal with these constraints, approaches have been proposed recently (Marijan et al., 2013; Marijan, 2015; Marijan et al., 2017, 2019; Xiao et al., 2018; Haghighatkhah et al., 2018; Busjaeger and Xie, 2016). Most of them are history-based, that is, they consider test cases that failed in the past are more likely to fail in the future. This kind of TCP technique has been acknowledged as one of the most suitable for the CI environment characteristics (Haghighatkhah et al., 2018).

Nevertheless, these approaches present some limitations. Some of them require code analysis, which can be costly. They do not consider test case volatility, a characteristic associated with the fact that test cases may be added and/or removed over the cycles. They are not adaptive, that is, they do not learn with past prioritizations. To overcome such limitations, learning approaches based on historical failure data have been proposed (Prado Lima and Vergilio, 2020a; Spieker et al., 2017). These approaches observe previous cycles and learn with past prioritizations guided by a reward function (online learning). We can mention two learning approaches that have produced promising results in TCP in CI environments (TCPCI) considering standard TCP metrics, such as Normalized Average Percentage of Faults Detected (NAPFD) and Average Percentage of Faults Detected with cost consideration (APFDc) (Spieker et al., 2017; Prado Lima and Vergilio, 2020a): i) RETECS (Spieker et al., 2017) (Reinforced Test Case Selection), which uses Reinforcement Learning (RL) to perform the prioritization; and ii) COLEMAN (Prado Lima and Vergilio, 2020a) (Combinatorial VOlatiLE Multi-Armed BANdit), an approach that uses a Multi-Armed Bandit (MAB) policy.

These two learning-based approaches are the focus of our work. They deal properly with the test case volatility and CI constraints and spend a time to perform the prioritization that is suitable for the CI budget. In a previous work we compared COLEMAN against a search-based approach. As a result, the learning-based approach presented solutions for most systems that are very near to the solutions produced by a Genetic Algorithm, considering early fault detection. Moreover, in return, there is a considerable gain in the time, mainly for the hard cases.

However, there are many difficulties inherent to the TCPCI problem, and we do not get there yet. Sure there is room for improvement. With this in mind, and following up our previous works, this paper aims to evaluate how far the solutions produced by both learning-based approaches are from optimal solutions. To answer this question, we evaluate the approaches having a deterministic approach as a baseline (ground truth). We adopted the same methodology of our previous work Prado Lima and Vergilio (2020b) and report results from the use of two reward functions: Reward Based on Failures and Reward Based on Time-Rank, in the presence of three different time constraints (budgets). But, differently, in the present work, we use twelve large-scale real-world software systems and six measures to evaluate: fault detection effectiveness, early fault detection, test time reduction in the CI cycles, prioritization time, and accuracy, regarding the distance from the optimal solution.

In this way, we evaluate both learning approach solutions' trade-offs regarding fault detection and prioritization time. This allows the assessment of lightweight test case prioritization approaches in CI environments by evaluating how far their solutions are from optimal ones. As a consequence, we obtained new findings and discuss some implications which were not presented in the previous works, regarding: i) application of the approaches: we present guidelines that include the choice of the reward function, cost consideration regarding test case duration, and characteristics of the systems and budgets; ii) identification of limitations and possible improvements: we analyze some aspects regarding the number of test cases, failures distribution over test cases and CI cycles. Such aspects are drawbacks for the learning approaches and constitute gaps for directing future research; and iii) benchmark construction: we identify hard prioritization cases. In addition to this, we make available a replication package containing supplementary material[1].

The paper is structured as follows. Section 2 contains background about CI environments, TCP works for CI, and related work. Section 3.1 details evaluated approaches. Section 4 describes how our evaluation was conducted: objectives, Systems Under Test (SUT), evaluation measures and used parameters. Section 5 shows and analyses the results. Section 6 presents the main threats to the validity of our results. Section 7 contains our final remarks and discusses future work.

2 Background and Related Work

Continuous Integration (CI) environments have been increasingly adopted in the industry (Hilton, 2016). CI environments automate the process of building and testing software, and allow engineers to merge code that is under development or maintenance with the mainline codebase at frequent time intervals (Duvall et al., 2007; Fowler, 2006). In CI development, teams work continuously integrating code and make smaller code commits every day, usually monitored by a CI server. When a change occurs, the CI server clones this code, builds it, and runs the testing processes. When the entire process is finished, the CI server generates a report (feedback), and the developers are informed. Figure 1 illustrates this process.

2.1 TCP in CI environments

Given a test case set T, available for a build, the set PT of all possible permutations of T, and a function f that determines the performance of a given prioritization T' from PT to real numbers, the TCPCI problem aims at finding the best T' to achieve certain specific criteria measured by f. In CI, the determination of T' may subject to a test budget that is the available time to execute the CI cycle.

Many TCP approaches exist in the literature (Khatibsyarbini et al., 2018; Yoo and Harman, 2012). Among them, we can mention approaches that use evolutionary algorithms (Bajaj and Sangwan, 2019; Li et al., 2007; Di Nucci et al., 2018; Epitropakis et al., 2015). However, such approaches usually are complex and take much time to execute. In addition to this, most of them require coverage and code changes analysis, and do not consider the CI particularities such as volatility of test cases, multiple test requests, constraints and test budget. Because of this, approaches specific for CI have been proposed recently (Prado Lima and Vergilio, 2020c).

Marijan et al. (2013) introduce an approach named ROCKET. Such an approach implements domain specific heuristics that consider the distance of the failure

[1]Our supplementary material is available online (Prado Lima and Vergilio, 2021). After publication, we will use a DOI as a reference for the material.

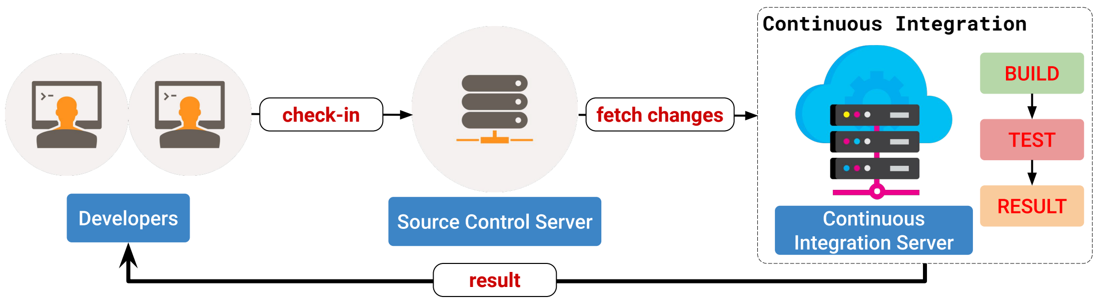


Figure 1. Overview of a Continuous Integration environment (Prado Lima and Vergilio, 2020a).

status from a current execution of a test case to its execution time. However, ROCKET does not consider the entire history of failures. An extension is proposed (Marijan, 2015) to consider, given a test budget, other fault detection, business, performance, and technical perspectives. Such an extension needs additional information related to coverage and features.

The tool, called TITAN (Marijan et al., 2017), uses constraint programming to minimize the number of test cases that cover some requirements that the original test cases cover. Then, the minimized set is prioritized using ROCKET (Marijan et al., 2013). The problem with these works is the cost because feature coverage or other additional information is required.

In the context of Highly Configuration Systems, Marijan et al. (2019) use a based-tree learning algorithm to classify test cases according to the feature coverage into the following categories: unique, totally redundant, and partially redundant. The main idea is to eliminate test cases totally redundant. Partially redundant test cases can also be included in the set according to its priority and time budget. The priority is calculated based on its historical fault detection effectiveness. Nevertheless, an effort to calculate the coverage is also necessary.

The work of Xiao et al. (2018) determines a priority for the test cases in the same commit and after, orders them considering the failure history, test coverage, test size, and execution time. The technique focuses only on test cases that failed recently, not exploring new test cases. Haghighatkhah et al. (2018) show the use of historical failure knowledge is a strong predictor for TCP in CI environments, and that it is effective to catch regression faults earlier without requiring a large amount of historical data. In addition to this, the effectiveness can be improved by using such knowledge with a diversity measure, calculated by comparing the text from test cases. The idea is not to calculate similarity based on measures that rely on code such as coverage, call methods, and so on.

Busjaeger and Xie (2016) use SVM^{map} to create a prediction model for the fault-proneness of test cases to be used in the prioritization, taking into account five attributes: test coverage of modified code, the textual similarity between tests and changes, recent test-failure or fault history, and test age. However, these attributes need additional information and rely on code instrumentation.

The works mentioned above present some limitations. They require code analysis, which can be costly. They do not address the main characteristics of CI environments. For instance, they do not consider test case volatility, a characteristic associated with the fact that test cases may be added and/or removed over the cycles. They are not adaptive, that is, they do not learn with past prioritizations.

To overcome such limitations, learning approaches based on historical failure data have been proposed. Bertolino et al. (2020) distinguish two kinds of TCP learning-based strategies. The first one, named Learning-to-Rank, uses supervised learning to train a model based on some test features. The model is then used to rank test sets in future commits. The problem with these strategies is that the model may no longer be representative when the commit context changes. The second kind, named Ranking-to-Learn, is more suitable to the dynamic CI context. This strategy learns based on the rewards obtained from the feedback of previously used ranks. The main idea is to maximize the rewards.

The work of Bertolino et al. (2020) presents results comparing both kinds of approaches in CI and evaluates the performance of different Machine Learning (ML) algorithms. They conclude that Ranking-to-Learn strategies are more robust regarding test case volatility, code changes, and number of failing tests. Because of this, the focus of our work is on this kind of strategy, evaluating two approaches that have presented promising results in TCPCI context: RETECS (Spieker et al., 2017) (Reinforced Test Case Selection) and COLEMAN (Prado Lima and Vergilio, 2020a) (Combinatorial VOlatiLE Multi-Armed BANdit).

In this sense, the work of Bertolino et al. has goals similar to ours. However, the evaluation conducted by that work uses a different approach, that is, in fact, a test case selection and prioritization approach. It includes a step to first select a test case subset before the prioritization through the learning strategies. In this way, the Ranking-to-Learn approaches were not evaluated as they were proposed in the literature. The work does not evaluate a MAB strategy, as used by COLEMAN. The learning is based on different features related to the test activity, which can be costly. We also used another set of evaluation measures, adopted in the test case prioritization context, leading to new findings and insights about the learning approaches. The next section describes both approaches RETECS and COLEMAN, in details.

3 Learning-based Approaches

This work aims to evaluate Ranking-to-Learn approaches for TCPCI problem by comparing their solutions with optimal solutions produced by a deterministic approach. Figure 2 illustrates how such approaches work in the CI environment. After a successful build, the approaches are applied before the test execution and perform the prioritization of a test case set (T) available for the current commit (c). Thus, the test cases of the prioritized test case set (T') are executed until the available test budget is reached. Feedback (reward) about T' execution is obtained and used the approaches to adapt its experience for future actions (online learning). In the end, a report is generated, and the developers are informed.

Next, we present the two learning-based approaches used in this work: RETECS and COLEMAN, as well as the reward functions used by both approaches in our evaluation, RNFail and TimeRank.

3.1 RETECS

RETECS (Reinforced Test Case Selection), introduced by Spieker et al. (2017), is a Reinforcement Learning (RL) based approach. It uses an agent, for instance, an Artificial Neural Network (ANN) or a Tableau Representation, to interact with the CI environment. Based on the environment state, the agent defines an action (prioritization) to be applied in such an environment. The state is given by the information about a test case, such as the test case duration, historical failure data, and previous last execution. After, according to its previous action's performance, the agent receives a reward (feedback).

Based on rewards provided by a reward function, the agent adapts its experience for future actions (online learning). To avoid learning with irrelevant information due to long history information (low reliability with the actual behavior of the system under test), a memory representation (sliding window) is used to delimit how much past information is used to learn.

Spieker et al. compared different variants of RL agents and evaluated the best variation against three basic TCP methods, a random prioritization and two deterministic methods: Sorting and Weighting. The Sorting method prioritizes giving higher priority for the test cases that failed recently, while Weighting is based on the weighted sum given to the input information used as state in the RL agent. According to the authors, RETECS using the ANN variant presented the best results. For this, we evaluated the performance of RETECS using an ANN as an agent in our study.

3.2 COLEMAN

COLEMAN is a Multi-Armed Bandit (MAB) (Kuleshov and Precup, 2014) based approach design to solve the TCPCI problem dealing with the Exploration versus Exploitation (EvE) dilemma (Robbins, 1985). In such a dilemma, there is a balance in the search between solutions with the best performance and dissimilar solutions.

In the MAB scenario, a player plays on a set of slot machines (or arms/actions) that even identical produce different gains. After a player pulls one of the arms in a turn (c), a reward is received drawn from some unknown distribution, thus aiming to maximize the sum of the rewards. Different strategies, called MAB policies, can be used to choose the next arm by observing previous rewards and decisions.

Similarly, COLEMAN considers that a test case is an arm, but it encompasses the dynamic nature of the TCPCI problem. For this, COLEMAN incorporates two variants of MAB: volatile and combinatorial. In the first variation, the approach selects multiple arms in each turn (commit), rather than one, to produce an ordered set. In the second one, only the test cases available in each commit are considered for prioritization. The second variation aims to deal with the test case volatility. In the end, a reward function is used to obtain feedback (reward) from the prioritization proposed by the approach. Based on such feedback, the approach aims to incorporate the learning from the application of the prioritized test case set.

COLEMAN is generic and lightweight. That is, it does not require any further detail about the system under tests such as code coverage or code instrumentation, as well as it allows the use of any MAB policy and requires only historical failure data. According to the authors of COLEMAN, it is possible to use different MAB policies. Among the policies evaluated in the experiments performed, the Fitness-Rate-Rank based on Multi-Armed Bandit (FRRMAB) policy (Li et al., 2014) presented the best performance.

FRRMAB is a state policy that works with a sliding window SW as a smoother way to consider dynamic environments, allowing the observation of the changes in the quality of the arms (test cases) along the search process. The use of a SW allows evaluating a test case without it being hampered by its performance at a very early stage, which may be irrelevant to its current performance.

The FRRMAB policy was used in a further study that analyses the trade-offs of the COLEMAN solutions in comparison with the near-optimal solutions generated by a Genetic Algorithm (GA) (Prado Lima and Vergilio, 2020b). Such an study shows that, except for one system, COLEMAN yields near-optimal solutions with negligible time. The unique exception was under a restrictive test budget associated with a few historical data. Because of this, FRRMAB is also adopted in our study that compares COLEMAN with a deterministic approach.

3.3 Reward Functions

Reward functions are used to evaluate the performance of a prioritization. In this work, we use the same functions adopted in (Prado Lima and Vergilio, 2020a).

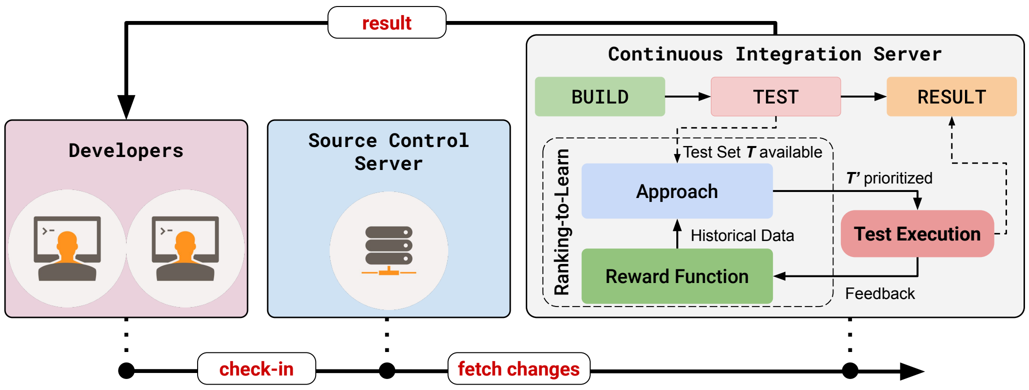


Figure 2. Integration of the evaluated learning approaches in the CI environment.

They are adapted from the work of Spieker et al. (2017). They are: Reward Based on Failures (RNFail) and Reward based on Time-ranked (TimeRank), as follows.

Let t'_c to be a test case from a prioritized test case set T'_c at a commit (cycle) c. The first reward function $RNFail$ (Equation 1) is based on the number of failures associated with a test case $t'_c \in T'_t$. This function captures the ability of a test case to produce failures.

$$RNFail(t'_c) = \begin{cases} 1 & \text{if } t'_c \text{ failed} \\ 0 & otherwise \end{cases} \tag{1}$$

The second function is TimeRank, defined in Equation 2. Let T'^{fail} is composed by the failing test cases from T'_c; The $prec(t'_{c_1}, t'_{c_2})$ function returns 1 if the position in T'_c of t'_{c_1} is lower than the position of t'_{c_2}.

$$TimeRank(t'_c) = \frac{|T'^{fail}| - [\neg(fails(t'_c))}{|T'^{fail}|} \times \frac{\sum_{i=1}^{|T'^{fail}|} prec(t'_c, t'_{c_i})]}{|T'^{fail}|} \tag{2}$$

The TimeRank function observes the rank of each t'_c in T'_c and considers the problem of early scheduling. This function privileges the failing test cases ranked in the first positions in T'_c, and it penalizes those that do not fail and precede failing ones. The reward value of non-failed test case is decreased according to the number of failing test cases ranked after it.

4 Evaluation Methodology

This section describes how the evaluation was conducted, by presenting objectives, used measures and systems, how the optimal solutions were obtained, and how RETECS and COLEMAN were executed.

The evaluation was guided by the following general research question: "How far are the solutions produced by learning approaches from optimal solutions obtained through a deterministic approach?" From this general question, we derived five objective research questions. Each one of these research questions evaluates the learning approaches considering a different perspective in comparison with the deterministic approach by using distinct measures as follows.

- RQ1: fault detection effectiveness.
- RQ2: fault detection effectiveness in the presence of the factor cost.
- RQ3: early fault detection and time reduction considering the CI cycles.
- RQ4: time the learning approaches take to perform the prioritization.
- RQ5: accuracy in the prediction considering the difference between the estimated values and the ones obtained by the deterministic approach.

To answer our question, we adopted a methodology following principles adopted by Wohlin et al. (2000). We formulated our main objectives according to the Goal Question Metric (GQM) method (Basili et al., 1994), as described in Table 1.

Table 1. Goal Question Metric Formulation

Goal:	to evaluate learning-based approaches
Purpose:	by analyzing their solutions
With respect to:	the optimal solutions generated by a deterministic approach
From the point of view:	of the tester
In the context of:	CI environments
Research Questions:	Evaluation Measures:
RQ1	fault detection effectiveness (NAPFD)
	fault detection effectiveness with cost consideration (APFDc)
RQ2	early fault detection (RFTC)
	test time reduction (NTR)
RQ3	prioritization time (PR)
RQ4	accuracy (RMSE)

4.1 Evaluation Measures

In our study, we used six measures. These metrics were chosen because they are largely used in the TCP literature (Yoo and Harman, 2012). The first one, NAPFD (Normalized APFD) metric (Qu et al., 2007) (Equation 3), evaluates fault detection effectiveness and is an extension of the APFD Average Percentage of Faults Detected (APFD) (Rothermel et al., 1999) metric. APFD indicates how quickly a set of prioritized test cases (T') can detect faults present in the application being tested. On the other hand, the NAPFD metric considers the ratio between detected and detectable faults within T. NAPFD fits the CI time constraints adequately, when not all the test cases are executed due to a time budget, and faults can be undetected. Higher NAPFD values indicate that the faults are detected faster using fewer test cases.

$$NAPFD(T') = p - \frac{\sum_1^n rank(T'_i)}{m \times n} + \frac{p}{2n} \tag{3}$$

where m denotes the number of faults detected by all test cases; $rank(T'_i)$ is the position of T'_i in T', if T'_i did not reveal a fault we set $T'_i = 0$; n denotes the number of test cases in T'; and p denotes the number of faults detected by T' divided by m. NAPFD is equal to APFD metric if all faults are detected.

The second measure, named APFDc (APFD with cost consideration) (Elbaum et al., 2001) (Equation 4), is also an extension from APFD. This metric assumes that the test cases do not have the same cost. Thus, we can consider that a test case can be more costly to execute than others, concerning, for instance, to execution time. The cost can be used as a limit, in which the test cases are usually prioritized until a maximum cost is reached. Besides that, APFDc can compute the APFD value, whether both fault severity and test case costs are identical. In this work, we consider that all faults have the same severity.

$$APFDc(T'_t) = \frac{\sum_{i=1}^m (\sum_{j=TF_i}^n c_j - 0.5c_{TF_i})}{\sum_{j=1}^n c_j \times m} \tag{4}$$

where c_j is the cost of a test case T_i, and TF_i is the first test case from T' that reveals fault i.

The third measure Rank of the Failing Test Cases (RFTC) evaluates the test suite efficiency concerning early fault detection. In such a rank, the value represents the first failing test cases' execution order in the prioritized test suite.

We defined a fourth measure, Normalized Time Reduction (NTR) (Equation 5), to capture the difference between the time spent until the first test case fails r_t and the total time spent to execute all tests $\hat{r}_t$. Only the commits which failed CI^{fail} are considered in the calculation. In this way, we can evaluate the capability of an algorithm to reduce the time spent in a CI cycle.

$$NTR(\mathcal{A}) = \frac{\sum_{t=1}^{CI^{fail}} (\hat{r}_t - r_t)}{\sum_{t=1}^{CI^{fail}} (\hat{r}_t)} \tag{5}$$

The fifth measure, Prioritization Time (PR), measures the time spent by an approach to perform the prioritization. PR evaluates the applicability of an approach. Based on this value, we can observe whether an approach spends much time, making it impracticable for real scenarios.

The last measure, Root-Mean-Square-Error (RMSE), measures the difference between the predicted and the observed values of NAPFD (or APFDc). In our case, RMSE (Equation 6) is the difference between the value calculated for T' in a CI Cycle (commit) t, suggested by the learning approaches ($\hat{s}_t$) and the optimal value T' (s_t) found by the deterministic approach. For an algorithm $\mathcal{A}$, the RMSE is computed as follows:

$$RMSE(\mathcal{A}) = \sqrt{\frac{\sum_{t=1}^{CI} (\hat{s}_t - s_t)^2}{CI}} \tag{6}$$

where CI is the amount of CI Cycles in a system. The most accurate approach is the one with smallest RMSE.

We compared the results using Kruskal-Wallis (Kruskal and Wallis, 1952) and Mann-Whitney (Mann and Whitney, 1947) statistical tests to determine the significance level, and Vargha and Delaney's $\hat{A}_{12}$ (Vargha and Delaney, 2000) metric as effect test. The statistical tests were applied with 95% of confidence. The $\hat{A}_{12}$ metric calculates the effect size magnitude of the difference between two groups, which defines the probability of a value taken randomly from the first sample is higher than a value taken randomly from the second sample.

The magnitude can be: *Negligible* ($\hat{A}_{12} < 0.56$); *Small* ($0.56 \leq \hat{A}_{12} < 0.64$); *Medium* ($0.64 \leq \hat{A}_{12} < 0.71$); and *Large* ($0.71 \leq \hat{A}_{12}$). A *Negligible* magnitude represents a very small difference among the values and usually does not yield statistical difference. The *Small* and *Medium* magnitudes may yield statistical differences (or not). Finally, a *Large* magnitude represents a significantly large difference that usually can be seen in the numbers without much effort.

A *Negligible* magnitude represents a very small difference among the values and usually does not yield statistical difference. The *Small* and *Medium* magnitudes represent small and medium differences among the values, and may or not yield statistical differences. Finally, a *Large* magnitude represents a significantly large difference that usually can be seen in the numbers without much effort.

4.2 Target Systems

The study was performed with twelve systems already used in the literature (Prado Lima and Vergilio, 2020a; Spieker et al., 2017; Yu et al., 2019). The target systems are detailed in Table 2 that contains: the system name, the period of build logs analyzed, the total of builds identified, and in parentheses, the number of builds included in the analysis. Build logs with some problems were discarded, e.g., extracting information (non-valid build log), and those the test cases did not execute.

The fourth column shows the total of failures found; in parentheses, the number of builds in which at least one test failed. The fifth column shows the number of unique test cases identified from build logs; in parenthesis, the range of test cases executed in the builds. The sixth and seventh columns present the average duration and standard deviation in minutes of the CI Cycles (commits), and the interval between them.

Druid, developed by Alibaba, is a database connection pool written in Java. Fastjson, created by Alibaba, is a Java library that can be used to a fast JSON parser/generator for Java. Deeplearning4j is a deep learning library for Java Virtual Machine. DSpace is an open source software that provides facilities for the management of digital collections, used for the implementation of institutional repositories. GSDTSR is The Google Dataset of Testing Results (Elbaum et al., 2014) with a sample of 3.5 million test suite execution results from Google products. Guava, developed by Google, is a set of core libraries for Java which includes new collection types, APIs/utilities for concurrency, I/O, and others. OkHttp, developed by Square, is an HTTP and HTTP/2 client for Android and Java applications. Retrofit, also developed by Square, is a type-safe HTTP client for Android and Java. ZXing (Zebra Crossing) is a barcode scanning library for Java and Android. The systems IOF/ROL and Paint Control are industrial datasets for testing complex industrial robots from ABB Robotic (Spieker et al., 2017). LexisNexis is an industrial dataset for testing complex web-system at LexisNexis company (Yu et al., 2019).

More details about these systems are available in our replication package (Prado Lima and Vergilio, 2021). This package contains some figures illustrating number of failures per cycle for each SUT, which allows observing test case volatility.

4.3 Generating Optimal Solutions

The deterministic approach finds the optimal solution for each commit associated with a SUT. The failure results from the test case execution are known a priori and used in the prioritization, according to Algorithm 1.

We identify the test set available T_c and the original time A_c spent to run such set in each commit. After, we define a time budget to run the tests. In this work, we evaluated three time budgets (TB_c) concerning, respectively, 10%, 50%, and 80% of the execution time of the overall test set T_c available. They were chosen to observe the influence of the test budget in the results and how it affects the learning process, as well as they already were used in previous work (Prado Lima and Vergilio, 2020a). In the end, we sort the test case by the number of failures in descending order, and test case duration in ascending order. This sorter allows evaluating the prioritized test set through different measures, such as failure detection (NAPFD) and cost (APFDc).

Algorithm 1: Deterministic Algorithm to Find Optimal Solutions for Test Case Prioritization in Continuous Integration Environments Problem.

```
forall commit $c$ in Target System do
    $T_c \leftarrow$ Test Case set available from system in the current commit;
    $A_c \leftarrow$ Total time spent to run $T_c$;
    $TB_c \leftarrow$ Time Budget (10%, 50%, or 80% from $A_c$);
    $T'_c \leftarrow T_c$ ordered by number of failures (descending) and duration (ascending);
    Evaluate $T'_c$ considering $TB_c$ (e.g. NAPFD and APFDc);
end
```

4.4 Executing Learning-based approaches

We use the results from the execution of COLEMAN and RETECS available in (Prado Lima and Vergilio, 2020a)[2]. They were obtained with 30 independent executions for each algorithm/configuration, using both reward functions, RNFail and TimeRank. COLEMAN was configured with FRRMAB policy, sliding window size SW equals to 100, coefficient C to balance exploration and exploitation equals to 0.3, and decayed factor equals to 1. RETECS was executed with an Artificial Neural Network (ANN), and the values used for Hidden Nodes, Replay Memory, and Replay Batch Size, are, respectively, 12, 10000, and 1000. All the experiments were performed on an Intel Xeon E5-2640 v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS. The system LexisNexis was not used in previous work (Prado Lima and Vergilio, 2020a). For this system, we executed the experiments following the same settings abovementioned. The deterministic algorithm was executed in the same computational environment.

5 Results and Analysis

In this section, we analyze the results of the learning approaches, RETECS and COLEMAN, having as a baseline the deterministic approach and our measures. Each subsection evaluates a different perspective according the research questions posed previously.

5.1 RQ1: Fault Detection Effectiveness

To evaluate the prioritization quality regarding fault detection capacity, we compare NAPFD average values presented in Table 3. This table presents average values ± standard deviation. The best values are highlighted in bold. We applied the Kruskal-Wallis test to compare the algorithms regarding each measure. Results that are statistically equivalent to the best one are highlight in gray. We also use different symbols to indicate the effect size magnitude concerning the best values: "★" denotes the best algorithm for a time budget in a SUT. "▼"

[2]Supplementary material available at `https://doi.org/10.17605/OSF.IO/WMCBT`

Table 2. Description of the Target Systems

Name	Period	Builds	Failures	Test Cases	Duration (min)	Interval (min)
Druid	2016/04/24-2016/11/08	286 (168)	270 (71)	2391 (1778-1910)	4.97 ± 10.66	384.76 ± 468.86
Fastjson	2016/04/15-2018/12/04	2710 (2371)	940 (323)	2416 (900-2102)	1.97 ± 0.89	233.22 ± 401.26
Deeplearning4j	2014/02/22-2016/01/01	3410 (483)	777 (323)	117 (1-52)	12.33 ± 14.91	306.05 ± 442.55
DSpace	2013/10/16-2019/01/08	6309 (5673)	13413 (387)	211 (16-136)	11.78 ± 7.03	291.29 ± 411.19
GSDTSR	2016/01/02-2016/02/01	259388 (259388)	3208 (2924)	5555 (1-390)	974.25 ± 4850.66	1439.91 ± 2.58
Guava	2014/11/06-2018/12/02	2011 (1689)	7659 (112)	568 (308-512)	62.53 ± 80.31	435.55 ± 464.52
IOF/ROL	2015/02/13-2016/10/25	2392 (2392)	9289 (1627)	1941 (1-707)	1537.27 ± 2018.73	1324.36 ± 291.78
LexisNexis	2018/09/27-2018/11/15	54 (54)	21189 (54)	2662 (2007-2377)	0.8668 ± 0.808	900.367 ± 305.125
OkHttp	2013/03/26-2018/05/30	9919 (6215)	9586 (1408)	289 (2-75)	7.64 ± 5.64	220.17 ± 405.93
Paint Control	2016/01/12-2016/12/20	20711 (20711)	4956 (1980)	1980 (1-74)	424.46 ± 275.90	1417.86 ± 144.97
Retrofit	2013/02/17-2018/11/26	3719 (2711)	611 (125)	206 (5-75)	2.40 ± 1.60	270.86 ± 449.41
ZXing	2014/01/17-2017/04/16	961 (605)	68 (11)	124 (81-123)	13.14 ± 12.37	411.10 ± 465.53

indicates a negligible effect size; "▽" denotes a small magnitude, "△" a medium magnitude, and "▲" large.

As expected, the deterministic approach presents the best values for all systems and budgets with statistical difference, that has, in most cases, a large magnitude.

Although there are statistical differences, in some cases, we observe statistical equivalence, mainly for COLEMAN using TimeRank function, in the less restrictive scenarios (budgets of 50% and 80%). This means the results are very close to optimal. Using TimeRank, for all systems and budgets, COLEMAN obtained equivalence to the optimal in 15 cases out of 36 (≈ 42%). Considering each budget 10%, 50%, and 80%, COLEMAN reaches equivalence in, respectively, 17%, 50%, and 58% of the cases. COLEMAN does not reach such a good performance using RNFail function. In contrast, RETECS reaches results equivalent to the optimal using RNFail function, but only in 3 cases, out of 36 (≈ 8%). Its performance seems not be impacted by the test budget. In conclusion, COLEMAN outperforms RETECS regarding early fault detection.

Finding 1. COLEMAN presents better performance than RETECS. Using TimeRank, COLEMAN obtained results that are equivalent to optimal in 42% of the cases. This percentage increases in the presence of less restrictive budgets, reaching 58% of the cases in the time budget of 80%.

To a better visualization, Figure 3 illustrates radar charts (or spider graphs) for each time budget. These charts allow us to observe the variation of the algorithms across the systems as well as the difference between them. Each angle represents the NAPFD value for a system. The purple line represents the values found by the Deterministic approach; blue and orange lines obtained by RETECS using, respectively, RNFail and TimeRank functions; and green and red lines obtained by COLEMAN using RNFail and TimeRank functions.

As we can observe, increasing the time budget, the learning-based approaches produce solutions closer to optimal, mainly COLEMAN. In some systems, we observe that, even increasing the time budget, the difference keeps the same; for instance, Deeplearning4j and OkHttp. To a deeper analysis, we refer to Figure 4. This figure presents, in overall (in the same scale), different information about each system: number of valid builds, number of failures, number of failed builds, number of test cases, mean number of failures by cycles, and mean number of failing cycles. More information is found in Table 2.

Regarding the Deeplearning4j system, it has a high average of failing builds. However, as we observed, only this does not help to provide good prioritization. On the other hand, OkHttp has a small average of failing builds but more failures and failed builds. This helped to provide better NAPFD values than in the Deepleaning4j system. In both systems, COLEMAN obtained equivalence to optimal in all time budgets evaluated.

We analyzed other characteristics of the systems that may impact the prioritization, e.g., the test case volatility. We observe in the Deeplearning4j and OkHttp systems that the failures are frequent and well distributed in some tests, even having peaks of failures and test case volatility. This scenario endorses an approach based on historical test data.

Among the systems, ZXing is the simplest one. In this system, the values found by learning-based approaches are the closest to the optimal, that is, close to 1 which is the maximum value for NAPFD. There is low test case volatility in this system, and there are peaks in the failure detection in a few commits, with long periods without failures. This situation can also be supported by an approach based on historical test data.

Regarding the NAPFD results that are equivalent to the optimal, we observe that RNFail fits better with RETECS and TimeRank with COLEMAN. The worst results were obtained by RETECS and COLEMAN in the Druid and LexisNexis systems. The Druid system presents the greatest difference between learning-based approaches and Deterministic, mainly in the presence of the most restrictive time budget (10%). These systems share some particularities: (i) a few number of CI Cycles; and (ii) a large test case set, in which many failures are distributed in many test cases. Apparently, such characteristics are drawbacks for approaches based on historical test data.

Table 3. NAPFD comparison.

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
Time Budget: 10%					
Druid	0.6768 ± 0.129 ▲	0.6488 ± 0.074 ▲	0.9996 ± 0.000 ★	0.6801 ± 0.052 ▲	0.7137 ± 0.074 ▲
Fastjson	0.8713 ± 0.015 ▲	0.8719 ± 0.005 ▲	0.9988 ± 0.000 ★	0.9030 ± 0.014 ▲	0.8980 ± 0.018 ▲
Deeplearning4j	0.6615 ± 0.072 ▲	0.6739 ± 0.016 ▲	0.8137 ± 0.000 ★	0.7533 ± 0.002 ▲	0.7716 ± 0.000 ▲
DSpace	0.9437 ± 0.001 ▲	0.9410 ± 0.001 ▲	0.9739 ± 0.000 ★	0.9489 ± 0.003 ▲	0.9496 ± 0.004 ▲
GSDTSR	0.9893 ± 0.000 ▲	0.9893 ± 0.000 ▲	0.9898 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9676 ± 0.015 ▲	0.9563 ± 0.004 ▲	0.9978 ± 0.000 ★	0.9554 ± 0.002 ▲	0.9586 ± 0.001 ▲
IOF/ROL	0.3704 ± 0.005 ▲	0.3779 ± 0.003 ▲	0.4098 ± 0.000 ★	0.3632 ± 0.001 ▲	0.3670 ± 0.001 ▲
LexisNexis	0.0508 ± 0.018 ▲	0.1004 ± 0.068 ▲	0.7011 ± 0.000 ★	0.1400 ± 0.001 ▲	0.1440 ± 0.001 ▲
Paint Control	0.9078 ± 0.000 ▼	0.9077 ± 0.000 ▲	0.9078 ± 0.000 ★	0.9076 ± 0.000 ▲	0.9076 ± 0.000 ▲
OkHttp	0.8357 ± 0.002 ▲	0.8095 ± 0.006 ▲	0.8886 ± 0.000 ★	0.8323 ± 0.000 ▲	0.8407 ± 0.000 ▲
Retrofit	0.9641 ± 0.001 ▲	0.9621 ± 0.001 ▲	0.9712 ± 0.000 ★	0.9639 ± 0.000 ▲	0.9642 ± 0.000 ▲
ZXing	0.9854 ± 0.000 ▲	0.9855 ± 0.000 ▲	0.9998 ± 0.000 ★	0.9826 ± 0.000 ▲	0.9828 ± 0.000 ▲
Time Budget: 50%					
Druid	0.6851 ± 0.134 ▲	0.6323 ± 0.074 ▲	0.9996 ± 0.000 ★	0.9333 ± 0.013 ▲	0.9710 ± 0.008 ▲
Fastjson	0.8714 ± 0.007 ▲	0.8902 ± 0.013 ▲	0.9993 ± 0.000 ★	0.9174 ± 0.021 ▲	0.9118 ± 0.028 ▲
Deeplearning4j	0.7049 ± 0.070 ▲	0.6562 ± 0.018 ▲	0.9025 ± 0.000 ★	0.7890 ± 0.001 ▲	0.8200 ± 0.000 ▲
DSpace	0.9568 ± 0.001 ▲	0.9485 ± 0.001 ▲	0.9921 ± 0.000 ★	0.9724 ± 0.009 ▲	0.9766 ± 0.008 ▲
GSDTSR	0.9911 ± 0.000 ▲	0.9906 ± 0.000 ▲	0.9921 ± 0.000 ★	0.9893 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9502 ± 0.015 ▲	0.9578 ± 0.004 ▲	0.9997 ± 0.000 ★	0.9653 ± 0.004 ▲	0.9675 ± 0.007 ▲
IOF/ROL	0.5101 ± 0.007 ▲	0.5025 ± 0.006 ▲	0.5812 ± 0.000 ★	0.5046 ± 0.002 ▲	0.5189 ± 0.002 ▲
LexisNexis	0.1629 ± 0.026 ▲	0.2335 ± 0.099 ▲	0.9065 ± 0.000 ★	0.5332 ± 0.007 ▲	0.5625 ± 0.008 ▲
Paint Control	0.9150 ± 0.000 ▲	0.9138 ± 0.000 ▲	0.9153 ± 0.000 ★	0.9150 ± 0.000 ▲	0.9150 ± 0.000 ▲
OkHttp	0.8812 ± 0.010 ▲	0.8446 ± 0.003 ▲	0.9544 ± 0.000 ★	0.9192 ± 0.000 ▲	0.9317 ± 0.000 ▲
Retrofit	0.9706 ± 0.002 ▲	0.9718 ± 0.002 ▲	0.9946 ± 0.000 ★	0.9853 ± 0.000 ▲	0.9893 ± 0.000 ▲
ZXing	0.9878 ± 0.000 ▲	0.9881 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9846 ± 0.000 ▲	0.9857 ± 0.000 ▲
Time Budget: 80%					
Druid	0.6490 ± 0.113 ▲	0.6551 ± 0.099 ▲	0.9996 ± 0.000 ★	0.938 ± 0.012 ▲	0.9830 ± 0.003 ▲
Fastjson	0.8708 ± 0.007 ▲	0.8925 ± 0.010 ▲	0.9999 ± 0.000 ★	0.9536 ± 0.010 ▲	0.9242 ± 0.028 ▲
Deeplearning4j	0.7058 ± 0.091 ▲	0.6640 ± 0.016 ▲	0.9520 ± 0.000 ★	0.8424 ± 0.001 ▲	0.8641 ± 0.001 ▲
DSpace	0.9601 ± 0.001 ▲	0.9508 ± 0.001 ▲	0.9932 ± 0.000 ★	0.9792 ± 0.006 ▲	0.9825 ± 0.007 ▲
GSDTSR	0.9921 ± 0.000 ▲	0.9914 ± 0.000 ▲	0.9934 ± 0.000 ★	0.9893 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9441 ± 0.012 ▲	0.9581 ± 0.007 ▲	0.9999 ± 0.000 ★	0.9784 ± 0.012 ▲	0.9841 ± 0.014 ▲
IOF/ROL	0.5495 ± 0.006 ▲	0.5287 ± 0.007 ▲	0.6115 ± 0.000 ★	0.5569 ± 0.002 ▲	0.5678 ± 0.001 ▲
LexisNexis	0.2496 ± 0.048 ▲	0.3545 ± 0.131 ▲	0.9152 ± 0.000 ★	0.6442 ± 0.005 ▲	0.7033 ± 0.004 ▲
Paint Control	0.9162 ± 0.000 ▲	0.9160 ± 0.000 ▲	0.9180 ± 0.000 ★	0.9171 ± 0.000 ▲	0.9171 ± 0.000 ▲
OkHttp	0.9027 ± 0.013 ▲	0.8558 ± 0.004 ▲	0.9607 ± 0.000 ★	0.935 ± 0.000 ▲	0.9478 ± 0.000 ▲
Retrofit	0.9724 ± 0.005 ▲	0.9745 ± 0.003 ▲	0.9972 ± 0.000 ★	0.9881 ± 0.000 ▲	0.9916 ± 0.000 ▲
ZXing	0.9878 ± 0.000 ▲	0.9883 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9972 ± 0.000 ▲	0.9996 ± 0.000 ▲

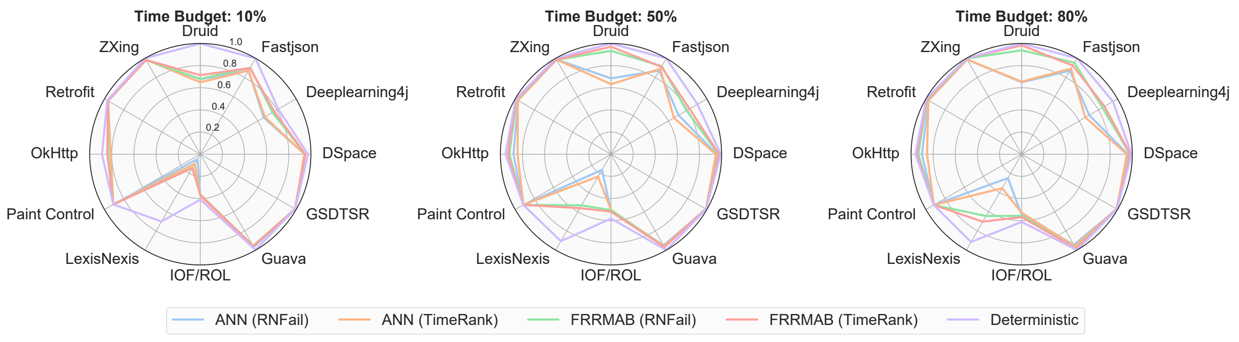
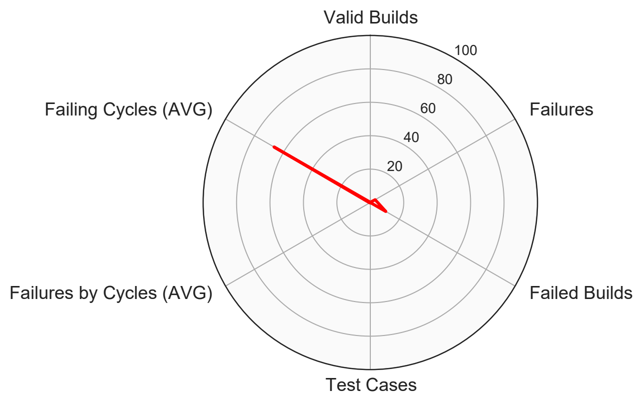


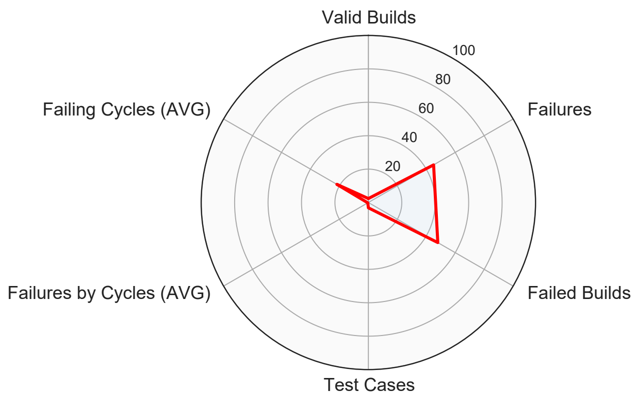
Figure 3. Radar charts - NAPFD values.

Finding 2. The learning-based approaches have the worst performance in systems with a high test case volatility and a few number of CI Cycles.

Concerning the optimal results, some of them are far from the maximum value for the metric, specifically for the systems IOF/ROL and LexisNexis. About the IOF/ROL system, we observe that the difficulty in obtaining better NAPFD values is related to the extremely high test case volatility coupled with the high number of failures, as well as the failure distribution over many test cases. This hampers to find a reasonable prioritization. On the other hand, in the LexisNexis system, we do not observe high test case volatility but, similarly to IOF/ROL, a high number of failures distributed in many test cases. In this way, both systems are examples of why it is hard to find reasonable solutions for TCP in the CI environments.



(a) Deeplearning4j



(b) OkHttp

Figure 4. Radar charts - systems characteristics

> Finding 3. A high number of failures distributed over many test cases makes the TCP task harder.

5.2 RQ2: Fault Detection Effectiveness with Cost Consideration

To evaluate how good is the prioritization considering the cost associated for each test case, we calculate APFDc values, presented in Table 4. We use the test case duration as cost. If two test cases reveal the same number of failures and have different execution times, that one that takes less time needs to appear first in the prioritization rank.

As it happens for NAPFD, COLEMAN performs better in the less restrictive budget using TimeRank, reaching results equivalent to the optimal in 50% of the cases for the time budget of 80%. RETECS also performs better using RNFail, but we observe a better performance of RETECS, overcoming COLEMAN, in the more restrictive budget of 10%. This is maybe due to the RETECS formulation that considers test case duration during its prioritizations. Nevertheless, considering a general case, COLEMAN outperforms RETECS, even only focusing on historical failure data.

> Finding 4. Regarding APFDc values COLEMAN outperforms RETECS in most cases. However, RETECS has a better performance in the most restrictive budget of 10%.

Again, radar charts regarding APFDc values can provide a better analysis (Figure 5). We can see that it is harder to obtain good prioritizations with less cost for LexisNexis for both approaches. We observe a great difference between COLEMAN and RETECS in the Druid system and the time budgets of 50% and 80%. As we observed in the NAPFD values, COLEMAN also has better performance with TimeRank function, whilst RETECS with RNFail. Besides that, NAPFD and APFDc values are not so far from optimal solutions, in which we can observe close values but with a statistical difference.

> Finding 5. The analysis of APFDc using test case duration as cost leads to results similar to those obtained in the NAFPD analysis. In general, the APFDc values are close to the optimal, and the more significant differences are obtained to the same systems.

5.3 RQ3 Early Fault Detection and Test Time Reduction

First of all, we calculate RFTC values (Table 5) that takes into account the position of the first failing test case in the prioritized test case set. We observed that the NAPFD average values found and the early fault detection (using RFTC) are correlated, that is, good NAPFD values provide good RFTC values. However, the opposite can not be true, once that the RFTC metric does not evaluate the prioritization quality from the entire prioritized test set but only the early fault detection.

As expected, the deterministic approach presents the best results for all systems. Besides that, COLEMAN using TimeRank obtained equivalent results in $\approx 70\%$ of the cases, whilst RETECS only in $\approx 3\%$ (only one case). In some cases, RETECS has a higher standard deviation than COLEMAN.

> Finding 6. Regarding RFTC, COLEMAN using TimeRank obtained performance equivalent to the optimal results in $\approx 70\%$ of the cases. COLEMAN is better than RETECS in all cases.

Early fault detection contributes to reduce test execution cost because the test can be ended when a failure occurs. Given this fact, we analyze NTR values (Table 6) to evaluate the impact in the time reduction inside a CI Cycle.

We can observe that in most cases, the greater the

Table 4. APFDc comparison.

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
			Time Budget: 10%		
Druid	0.8067 ± 0.088 ▲	0.7815 ± 0.051 ▲	0.9998 ± 0.000 ★	0.6964 ± 0.063 ▲	0.7181 ± 0.076 ▲
Fastjson	0.8805 ± 0.014 ▲	0.8810 ± 0.004 ▲	0.9985 ± 0.000 ★	0.9064 ± 0.013 ▲	0.8974 ± 0.018 ▲
Deeplearning4j	0.8135 ± 0.023 ▲	0.8185 ± 0.010 ▲	0.8304 ± 0.000 ★	0.7766 ± 0.001 ▲	0.7773 ± 0.000 ▲
DSpace	0.9480 ± 0.001 ▲	0.9458 ± 0.001 ▲	0.9724 ± 0.000 ★	0.9510 ± 0.002 ▲	0.9513 ± 0.004 ▲
GSDTSR	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲	0.9898 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9811 ± 0.007 ▲	0.9761 ± 0.003 ▲	0.9980 ± 0.000 ★	0.9561 ± 0.001 ▲	0.9582 ± 0.001 ▲
IOF/ROL	0.3746 ± 0.005 ▲	0.3819 ± 0.003 ▲	0.4138 ± 0.000 ★	0.3661 ± 0.001 ▲	0.3701 ± 0.001 ▲
LexisNexis	0.0541 ± 0.017 ▲	0.1025 ± 0.066 ▲	0.7076 ± 0.000 ★	0.1394 ± 0.001 ▲	0.1434 ± 0.001 ▲
Paint Control	0.9082 ± 0.000 ▲	0.9081 ± 0.000 ▲	0.9082 ± 0.000 ★	0.9080 ± 0.000 ▲	0.9080 ± 0.000 ▲
OkHttp	0.8484 ± 0.001 ▲	0.8292 ± 0.006 ▲	0.8870 ± 0.000 ★	0.8378 ± 0.000 ▲	0.8425 ± 0.000 ▲
Retrofit	0.9672 ± 0.001 ▲	0.9655 ± 0.001 ▲	0.9717 ± 0.000 ★	0.9646 ± 0.000 ▲	0.9648 ± 0.000 ▲
ZXing	0.9893 ± 0.000 ▲	0.9893 ± 0.000 ▲	0.9998 ± 0.000 ★	0.9832 ± 0.000 ▲	0.9835 ± 0.000 ▲
			Time Budget: 50%		
Druid	0.8147 ± 0.102 ▲	0.7597 ± 0.051 ▲	0.9998 ± 0.000 ★	0.9486 ± 0.016 ▲	0.9787 ± 0.009 ▲
Fastjson	0.9326 ± 0.005 ▲	0.9392 ± 0.017 ▲	0.9989 ± 0.000 ★	0.9186 ± 0.021 ▲	0.9140 ± 0.027 ▲
Deeplearning4j	0.8331 ± 0.041 ▲	0.8379 ± 0.012 ▲	0.9077 ± 0.000 ★	0.8106 ± 0.001 ▲	0.8134 ± 0.001 ▲
DSpace	0.9683 ± 0.001 ▲	0.9615 ± 0.001 ▲	0.9918 ± 0.000 ★	0.9737 ± 0.009 ▲	0.9767 ± 0.009 ▲
GSDTSR	0.9911 ± 0.000 ▲	0.9910 ± 0.000 ▲	0.9920 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9767 ± 0.008 ▲	0.9806 ± 0.005 ▲	0.9993 ± 0.000 ★	0.9687 ± 0.003 ▲	0.9672 ± 0.007 ▲
IOF/ROL	0.5175 ± 0.008 ▲	0.5043 ± 0.006 ▲	0.5905 ± 0.000 ★	0.5081 ± 0.002 ▲	0.5223 ± 0.002 ▲
LexisNexis	0.1891 ± 0.019 ▲	0.2477 ± 0.093 ▲	0.9035 ± 0.000 ★	0.5227 ± 0.007 ▲	0.5496 ± 0.007 ▲
Paint Control	0.9171 ± 0.000 ▲	0.9140 ± 0.000 ▲	0.9174 ± 0.000 ★	0.9162 ± 0.000 ▲	0.9162 ± 0.000 ▲
OkHttp	0.8878 ± 0.015 ▲	0.8869 ± 0.002 ▲	0.9477 ± 0.000 ★	0.9177 ± 0.000 ▲	0.9246 ± 0.000 ▲
Retrofit	0.9762 ± 0.002 ▲	0.9778 ± 0.002 ▲	0.9928 ± 0.000 ★	0.9850 ± 0.000 ▲	0.9885 ± 0.000 ▲
ZXing	0.9954 ± 0.000 ▲	0.9956 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9862 ± 0.000 ▲	0.9869 ± 0.000 ▲
			Time Budget: 80%		
Druid	0.7142 ± 0.111 ▲	0.6881 ± 0.090 ▲	0.9998 ± 0.000 ★	0.9469 ± 0.015 ▲	0.9912 ± 0.004 ▲
Fastjson	0.9037 ± 0.008 ▲	0.9133 ± 0.015 ▲	0.9991 ± 0.000 ★	0.9488 ± 0.012 ▲	0.9270 ± 0.026 ▲
Deeplearning4j	0.8158 ± 0.064 ▲	0.8522 ± 0.012 ▲	0.9407 ± 0.000 ★	0.8068 ± 0.002 ▲	0.7989 ± 0.001 ▲
DSpace	0.9738 ± 0.001 ▲	0.9639 ± 0.001 ▲	0.9925 ± 0.000 ★	0.9796 ± 0.006 ▲	0.9810 ± 0.008 ▲
GSDTSR	0.9919 ± 0.000 ▲	0.9917 ± 0.000 ▲	0.9930 ± 0.000 ★	0.9894 ± 0.000 ▲	0.9894 ± 0.000 ▲
Guava	0.9627 ± 0.010 ▲	0.9689 ± 0.009 ▲	0.9994 ± 0.000 ★	0.9780 ± 0.013 ▲	0.9825 ± 0.015 ▲
IOF/ROL	0.5593 ± 0.006 ▲	0.5311 ± 0.006 ▲	0.6225 ± 0.000 ★	0.5591 ± 0.002 ▲	0.5699 ± 0.001 ▲
LexisNexis	0.2886 ± 0.039 ▲	0.3569 ± 0.104 ▲	0.9111 ± 0.000 ★	0.6287 ± 0.005 ▲	0.6791 ± 0.004 ▲
Paint Control	0.9187 ± 0.000 ▲	0.9158 ± 0.000 ▲	0.9204 ± 0.000 ★	0.9176 ± 0.000 ▲	0.9177 ± 0.000 ▲
OkHttp	0.8836 ± 0.020 ▲	0.8974 ± 0.003 ▲	0.9520 ± 0.000 ★	0.9271 ± 0.000 ▲	0.9362 ± 0.000 ▲
Retrofit	0.9785 ± 0.004 ▲	0.9808 ± 0.003 ▲	0.9946 ± 0.000 ★	0.9873 ± 0.000 ▲	0.9903 ± 0.000 ▲
ZXing	0.9953 ± 0.000 ▲	0.9953 ± 0.001 ▲	0.9998 ± 0.000 ★	0.9975 ± 0.000 ▲	0.9996 ± 0.000 ▲

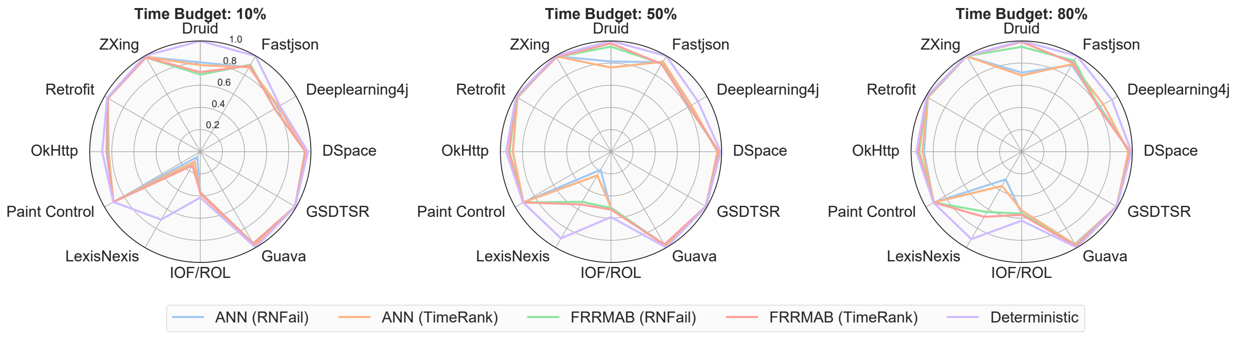


Figure 5. Radar charts - APFDc values.

time budget the greater the NTR values. The deterministic approach gets the best reduction values. However, other approaches have close values for most systems. The best percentages of time reduction for COLEMAN considering TimeRank function are in the systems: LexisNexis with 99.61% in all time budgets; and IOF/ROL with 57.01%, 71.93%, and 73.94%, for the time budgets, respectively of, 10%, 50%, and 80%. On the other hand, RETECS presents the best values considering RNFail function in Deeplearning4j, with 55.46%, 54.47%, and 54.75%, respectively for three budgets.

As mentioned before, for LexisNexis and IOF/ROL the failures are distributed over many test cases. This corroborates to the early fault detection once that there is a high probability of prioritizing a failing test case in the first positions.

The percentage found by the deterministic approach is low for some systems, such as Guava, Retrofit, and ZXing. In these systems, there is a low failure distribution across the test cases along with peaks of failures in a few CI Cycles, and the failing test cases vary in each CI Cycle. This shows that sometimes we face test cases that fail but spend much time executing, and there is not a pattern that hampers a reasonable prioritization

Table 5. RFTC comparison.

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
Time Budget: 10%					
Druid	1166.4411 ± 546.049 ▲	1240.8713 ± 356.817 ▲	1.0 ± 0.000 ★	209.3289 ± 98.987 ▲	56.1579 ± 31.056 ▲
Fastjson	1094.1147 ± 209.106 ▲	998.7949 ± 177.738 ▲	1.0 ± 0.000 ★	184.9911 ± 65.066 ▲	96.0378 ± 36.22 ▲
Deeplearning4j	5.3151 ± 2.456 ▲	5.7919 ± 0.677 ▲	1.0 ± 0.000 ★	2.3049 ± 0.048 ▲	2.0437 ± 0.007 ▲
DSpace	11.9561 ± 0.753 ▲	13.6276 ± 0.754 ▲	1.0 ± 0.000 ★	3.024 ± 1.33 ▲	2.1428 ± 0.67 ▲
GSDTSR	3.2553 ± 0.387 ▲	3.8958 ± 0.18 ▲	1.0 ± 0.000 ★	2.1316 ± 0.1 ▲	1.1482 ± 0.071 ▲
Guava	129.3477 ± 99.443 ▲	191.1219 ± 29.545 ▲	1.0 ± 0.000 ★	31.8991 ± 20.546 ▲	15.0977 ± 13.75 ▲
IOF/ROL	1.6445 ± 0.158 ▲	1.5639 ± 0.108 ▲	1.0 ± 0.000 ★	1.2626 ± 0.064 ▲	1.0992 ± 0.05 ▲
LexisNexis	73.097 ± 31.648 ▲	53.1777 ± 38.754 ▲	1.0 ± 0.000 ★	9.1333 ± 0.188 ▲	8.7611 ± 0.118 ▲
Paint Control	1.0 ± 0.000 ▼	1.001 ± 0.002 ▽	1.0 ± 0.000 ★	1.0 ± 0.000 ▼	1.0 ± 0.000 ▼
OkHttp	7.4112 ± 0.397 ▲	15.8935 ± 1.951 ▲	1.0 ± 0.000 ★	4.3424 ± 0.000 ▲	1.8039 ± 0.000 ▲
Retrofit	3.7352 ± 0.439 ▲	4.7297 ± 0.772 ▲	1.0 ± 0.000 ★	1.5152 ± 0.000 ▲	1.7941 ± 0.000 ▲
ZXing	39.7929 ± 0.643 ▲	39.1204 ± 1.701 ▲	1.0 ± 0.000 ★	1.0 ± 0.000 ▼	1.0 ± 0.000 ▼
Time Budget: 50%					
Druid	1225.6197 ± 600.746 ▲	1420.3143 ± 418.926 ▲	1.0 ± 0.000 ★	121.8396 ± 43.763 ▲	51.2697 ± 11.926 ▲
Fastjson	1527.9434 ± 93.004 ▲	1173.9871 ± 321.789 ▲	1.0 ± 0.000 ★	315.8923 ± 79.836 ▲	335.9929 ± 125.629 ▲
Deeplearning4j	5.0267 ± 1.827 ▲	7.3264 ± 0.53 ▲	1.0 ± 0.000 ★	2.5496 ± 0.011 ▲	2.464 ± 0.005 ▲
DSpace	19.0354 ± 0.887 ▲	27.6301 ± 0.921 ▲	1.0 ± 0.000 ★	5.5312 ± 1.939 ▲	4.1089 ± 1.769 ▲
GSDTSR	1.9072 ± 0.06 ▲	3.5648 ± 0.141 ▲	1.0 ± 0.000 ★	2.1461 ± 0.101 ▲	1.1505 ± 0.072 ▲
Guava	289.1586 ± 99.054 ▲	235.0919 ± 27.865 ▲	1.0 ± 0.000 ★	78.6409 ± 45.928 ▲	24.0869 ± 21.479 ▲
IOF/ROL	1.8009 ± 0.292 ▲	1.9213 ± 0.218 ▲	1.0 ± 0.000 ★	1.2588 ± 0.035 ▲	1.151 ± 0.024 ▲
LexisNexis	73.6444 ± 41.192 ▲	47.3883 ± 37.993 ▲	1.0 ± 0.000 ★	9.2074 ± 0.178 ▲	8.784 ± 0.091 ▲
Paint Control	1.0234 ± 0.004 ▲	1.0257 ± 0.007 ▲	1.0 ± 0.000 ★	1.0018 ± 0.001 ▲	1.0014 ± 0.001 ▲
OkHttp	6.203 ± 2.066 ▲	19.6306 ± 0.79 ▲	1.0 ± 0.000 ★	4.188 ± 0.014 ▲	2.3643 ± 0.002 ▲
Retrofit	5.4302 ± 0.43 ▲	5.625 ± 0.415 ▲	1.0 ± 0.000 ★	2.4059 ± 0.000 ▲	1.4299 ± 0.000 ▲
ZXing	51.2576 ± 0.579 ▲	49.4232 ± 3.15 ▲	1.0 ± 0.000 ★	5.6 ± 0.000 ▲	2.0 ± 0.000 ▲
Time Budget: 80%					
Druid	1427.8103 ± 493.429 ▲	1035.3369 ± 658.042 ▲	1.0 ± 0.000 ★	146.9112 ± 46.436 ▲	50.3805 ± 11.25 ▲
Fastjson	1535.2998 ± 101.625 ▲	993.382 ± 393.441 ▲	1.0 ± 0.000 ★	398.4345 ± 105.27 ▲	572.0954 ± 221.573 ▲
Deeplearning4j	5.5748 ± 2.358 ▲	7.8533 ± 0.581 ▲	1.0 ± 0.000 ★	2.8146 ± 0.014 ▲	2.5017 ± 0.011 ▲
DSpace	23.126 ± 1.021 ▲	33.4617 ± 1.119 ▲	1.0 ± 0.000 ★	6.8651 ± 2.946 ▲	6.0794 ± 3.343 ▲
GSDTSR	1.9413 ± 0.322 ▲	3.4858 ± 0.112 ▲	1.0 ± 0.000 ★	2.1461 ± 0.101 ▲	1.1505 ± 0.072 ▲
Guava	330.5342 ± 74.0 ▲	202.2301 ± 47.258 ▲	1.0 ± 0.000 ★	84.6856 ± 34.075 ▲	83.9989 ± 78.446 ▲
IOF/ROL	2.021 ± 0.447 ▲	2.5248 ± 0.362 ▲	1.0 ± 0.000 ★	1.317 ± 0.026 ▲	1.2366 ± 0.017 ▲
LexisNexis	47.6333 ± 43.432 ▲	43.5358 ± 39.71 ▲	1.0 ± 0.000 ★	9.2 ± 0.208 ▲	8.7981 ± 0.11 ▲
Paint Control	1.015 ± 0.002 ▲	1.0344 ± 0.009 ▲	1.0 ± 0.000 ★	1.0003 ± 0.001 ▽	1.0003 ± 0.001 ▽
OkHttp	4.2988 ± 2.242 ▲	21.5784 ± 1.148 ▲	1.0 ± 0.000 ★	4.0748 ± 0.021 ▲	2.282 ± 0.003 ▲
Retrofit	5.9252 ± 0.884 ▲	5.8832 ± 0.587 ▲	1.0 ± 0.000 ★	2.4636 ± 0.000 ▲	1.4386 ± 0.000 ▲
ZXing	51.0061 ± 1.233 ▲	48.6848 ± 2.926 ▲	1.0 ± 0.000 ★	4.2727 ± 0.000 ▲	1.3636 ± 0.000 ▲

using historical failure data.

Finding 7. Even using a deterministic approach, sometimes, the test time reduction is low due to peaks of failures, failure distributed across the test cases, and variation of the failing test cases over CI cycles. Nevertheless, COLEMAN reached high percentages of reductions for systems, considered hard cases for prioritizing, such as LexisNexis.

5.4 RQ4: Prioritization Time

We also observed the time spent to prioritize the test cases (Prioritization Time in Table 7). Although the deterministic approach is only a simple order, it can be used as a baseline.

We can observe the time spent by the approaches is negligible, even whimsy in most systems. A great time is spent in Druid, Fastjson, and LexisNexis systems that also have a significant number of test cases in a CI Cycle. RETECS using RNFail function has PR values that are statistically equivalent to the optimal in 23 (≈ 73%) cases out of 36. In three cases, it presents the best values for system Paint Control. But RETECS also presents the greatest variations; see system LexisNexis. In overall, RETECS and COLEMAN spend less than one second to perform the prioritization.

To observe the applicability in real scenarios, we considered the information presented in Table 2 regarding each SUT. In such a table, we present the time spent in a CI Cycle and the interval between commits for each system. As we can observe, typically, a new commit is performed, with a considered time, after a CI Cycle is ended. Such systems do not present a situation with multiple test requests, except in IOF/ROL system. As mentioned before, the approaches can reduce ≈ 99% of the CI Cycle time in such a system.

Moreover, the time presented in Table 2 is in minutes while the prioritization time of the approaches is presented in Table 7 in seconds. In this way, considering the interval between CI cycles, there is no negative impact in the use of the approaches. Furthermore, they can help developers concerning the time they spend waiting for test feedback.

Finding 8. The learning-based approaches are applicable in our real scenarios. Overall, they spend less than one second to execute.

Table 6. NTR comparison

SUT	RETECS		Deterministic	COLEMAN	
	RNFail	TimeRank		RNFail	TimeRank
Time Budget: 10%					
Druid	0.1863 ± 0.124	0.1556 ± 0.077	0.4331 ± 0.000	0.2027 ± 0.115	0.2355 ± 0.135
Fastjson	0.0194 ± 0.016	0.0219 ± 0.007	0.1442 ± 0.000	0.0681 ± 0.014	0.0566 ± 0.020
Deeplearning4j	0.5488 ± 0.029	0.5546 ± 0.015	0.5642 ± 0.000	0.4626 ± 0.001	0.4663 ± 0.000
DSpace	0.0188 ± 0.001	0.0105 ± 0.001	0.0476 ± 0.000	0.0239 ± 0.002	0.0251 ± 0.003
GSDTSR	0.0087 ± 0.000	0.0090 ± 0.000	0.0136 ± 0.000	0.0093 ± 0.000	0.0096 ± 0.000
Guava	0.0449 ± 0.015	0.0367 ± 0.005	0.0660 ± 0.000	0.0313 ± 0.001	0.0333 ± 0.002
IOF/ROL	0.5133 ± 0.030	0.5646 ± 0.015	0.6222 ± 0.000	0.5585 ± 0.007	0.5701 ± 0.004
LexisNexis	0.9784 ± 0.012	0.9863 ± 0.010	0.9999 ± 0.000	0.9960 ± 0.000	0.9961 ± 0.000
Paint Control	0.1151 ± 0.000	0.1139 ± 0.000	0.1152 ± 0.000	0.1133 ± 0.000	0.1131 ± 0.000
OkHttp	0.0830 ± 0.001	0.0579 ± 0.008	0.1160 ± 0.000	0.0658 ± 0.000	0.0702 ± 0.000
Retrofit	0.0088 ± 0.000	0.0076 ± 0.001	0.0126 ± 0.000	0.0070 ± 0.000	0.0073 ± 0.000
ZXing	0.0122 ± 0.000	0.0126 ± 0.001	0.0227 ± 0.000	0.0037 ± 0.000	0.0037 ± 0.000
Time Budget: 50%					
Druid	0.1840 ± 0.137	0.1213 ± 0.071	0.4331 ± 0.000	0.4057 ± 0.013	0.4225 ± 0.008
Fastjson	0.0399 ± 0.010	0.0602 ± 0.020	0.1445 ± 0.000	0.0768 ± 0.018	0.0724 ± 0.023
Deeplearning4j	0.5276 ± 0.039	0.5447 ± 0.010	0.5788 ± 0.000	0.4695 ± 0.000	0.4625 ± 0.000
DSpace	0.0334 ± 0.001	0.0218 ± 0.002	0.0604 ± 0.000	0.0486 ± 0.004	0.0499 ± 0.006
GSDTSR	0.0199 ± 0.000	0.0179 ± 0.000	0.0259 ± 0.000	0.0093 ± 0.000	0.0096 ± 0.000
Guava	0.0303 ± 0.016	0.0387 ± 0.006	0.0681 ± 0.000	0.0437 ± 0.002	0.0425 ± 0.005
IOF/ROL	0.7037 ± 0.019	0.6834 ± 0.014	0.7764 ± 0.000	0.7110 ± 0.003	0.7193 ± 0.003
LexisNexis	0.9894 ± 0.005	0.9902 ± 0.006	0.9999 ± 0.000	0.9959 ± 0.000	0.9961 ± 0.000
Paint Control	0.1283 ± 0.000	0.1142 ± 0.001	0.1290 ± 0.000	0.1222 ± 0.000	0.1223 ± 0.000
OkHttp	0.1118 ± 0.014	0.1060 ± 0.003	0.1671 ± 0.000	0.1431 ± 0.000	0.1486 ± 0.000
Retrofit	0.0134 ± 0.001	0.0138 ± 0.001	0.0188 ± 0.000	0.0156 ± 0.000	0.0172 ± 0.000
ZXing	0.0201 ± 0.000	0.0201 ± 0.001	0.0227 ± 0.000	0.0109 ± 0.000	0.0110 ± 0.000
Time Budget: 80%					
Druid	0.1477 ± 0.113	0.1230 ± 0.096	0.4331 ± 0.000	0.4069 ± 0.014	0.4292 ± 0.004
Fastjson	0.0385 ± 0.011	0.0516 ± 0.018	0.1445 ± 0.000	0.1040 ± 0.010	0.0860 ± 0.022
Deeplearning4j	0.5016 ± 0.058	0.5475 ± 0.007	0.5806 ± 0.000	0.4224 ± 0.001	0.4047 ± 0.000
DSpace	0.0374 ± 0.001	0.0269 ± 0.002	0.0606 ± 0.000	0.0525 ± 0.003	0.0526 ± 0.005
GSDTSR	0.0218 ± 0.001	0.0203 ± 0.000	0.0280 ± 0.000	0.0093 ± 0.000	0.0096 ± 0.000
Guava	0.0247 ± 0.013	0.0348 ± 0.009	0.0681 ± 0.000	0.0501 ± 0.012	0.0556 ± 0.011
IOF/ROL	0.7263 ± 0.015	0.6857 ± 0.011	0.7789 ± 0.000	0.7293 ± 0.002	0.7334 ± 0.001
LexisNexis	0.9913 ± 0.005	0.9902 ± 0.006	0.9999 ± 0.000	0.9959 ± 0.000	0.9961 ± 0.000
Paint Control	0.1285 ± 0.000	0.1161 ± 0.001	0.1310 ± 0.000	0.1209 ± 0.000	0.1204 ± 0.000
OkHttp	0.1112 ± 0.017	0.1153 ± 0.003	0.1674 ± 0.000	0.1493 ± 0.000	0.1551 ± 0.000
Retrofit	0.0140 ± 0.001	0.0145 ± 0.001	0.0195 ± 0.000	0.0161 ± 0.000	0.0179 ± 0.000
ZXing	0.0201 ± 0.000	0.0197 ± 0.002	0.0227 ± 0.000	0.0224 ± 0.000	0.0227 ± 0.000

5.5 RQ5: Accuracy

The accuracy (RMSE) is given by the difference between the predicted and the observed values of NAFPD and APFDc; these last ones are obtained by the deterministic approach. The results are presented in Tables 8 and 9. By analyzing such tables we can corroborate our previous findings.

Regarding the RMSE values for NAPFD metric (Table 8), we observe the predominance of COLEMAN (using TimeRank function) against RETECS. In the presence of a restrictive time budget (10%), RETECS performs better than in the other ones. However, COLEMAN is better in all time budgets.

The learning-based approaches obtain small RMSE values in almost all systems, except in Druid and Lexis Nexis. The smallest RMSE values are obtained under time budget of 80% and by COLEMAN for the systems Druid, DSpace, Guava, Paint Control, OkHttp, Retrofit, and ZXing. In these systems, the NAPFD values obtained are the closest to the optimal values. To a better visualization, we generated Figure 6.

One interesting point is that RMSE values lower than 0.2 represent NAPFD values closer to optimal values. For instance, in IOF/ROL system, the NAPFD values are low, but this is because it is challenging to find reasonable solutions. On the other hand, in the LexisNexis system, the deterministic approach also obtained low NAPFD values, but the learning-based approaches obtained the worst RMSE values. Such values are between 0.22 and 0.76.

RETECS gets the worst RMSE values for the systems Druid and LexisNexis in the time budget of 50%. They are, respectively, $RMSE \geq 0.58$ and $RMSE \geq\approx 0.75$. For these systems, this phenomenon occurs, as mentioned before, due to the lack of historical data. Besides them, we can observe in Figure 6 that the learning-based approaches also have a poor performance for Deeplearning4j. In this system, we do not find a correlation between test case volatility and the number of failures. For this, we investigated the accumulative NAPFD across the CI Cycles (Figure 7).

As we can observe, the NAPFD values change a bit before the 100th CI Cycle and normalize after the 300th. Near to the 100th cycle, the system Deeplearning4j starts presenting more failures, and the duration of some test cases starts increasing. Probably, such behavior influences more decisions taken by RETECS than by COLEMAN, once the first considers besides the failures, the test case duration. From then on, the number

Table 7. Prioritization Time (sec.) comparison.

SUT	RETECS RNFail	RETECS TimeRank	Deterministic	COLEMAN RNFail	COLEMAN TimeRank
			Time Budget: 10%		
Druid	0.3035 ± 0.031 ▲	0.3113 ± 0.026 ▲	0.0029 ± 0.000 ★	0.2383 ± 0.005 ▲	0.2316 ± 0.004 ▲
Fastjson	0.2108 ± 0.022 ▲	0.2025 ± 0.012 ▲	0.0029 ± 0.000 ★	0.4947 ± 0.193 ▲	0.4806 ± 0.191 ▲
Deeplearning4j	0.0037 ± 0.000 ▲	0.0179 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0272 ± 0.000 ▲	0.0272 ± 0.000 ▲
DSpace	0.0103 ± 0.001 ▲	0.0305 ± 0.002 ▲	0.0027 ± 0.000 ★	0.0296 ± 0.002 ▲	0.0296 ± 0.002 ▲
GSDTSR	0.0026 ± 0.000 ▲	0.0027 ± 0.000 ▲	0.0020 ± 0.000 ★	0.0253 ± 0.000 ▲	0.0253 ± 0.000 ▲
Guava	0.0335 ± 0.003 ▲	0.0385 ± 0.003 ▲	0.0027 ± 0.000 ★	0.0660 ± 0.013 ▲	0.0660 ± 0.013 ▲
IOF/ROL	0.0031 ± 0.000 ▲	0.2022 ± 0.036 ▲	0.0024 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
LexisNexis	0.4077 ± 0.036 ▲	0.5966 ± 0.052 ▲	0.0037 ± 0.000 ★	0.3328 ± 0.004 ▲	0.3277 ± 0.004 ▲
Paint Control	0.0019 ± 0.000 ★	0.0022 ± 0.000 ▲	0.0022 ± 0.000 ▲	0.0256 ± 0.000 ▲	0.0256 ± 0.000 ▲
OkHttp	0.0074 ± 0.001 ▲	0.0225 ± 0.002 ▲	0.0026 ± 0.000 ★	0.0286 ± 0.001 ▲	0.0285 ± 0.001 ▲
Retrofit	0.0044 ± 0.001 ▲	0.01 ± 0.002 ▲	0.0026 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
ZXing	0.0125 ± 0.001 ▲	0.0151 ± 0.001 ▲	0.0026 ± 0.000 ★	0.0343 ± 0.000 ▲	0.0342 ± 0.000 ▲
			Time Budget: 50%		
Druid	0.3881 ± 0.042 ▲	0.3844 ± 0.045 ▲	0.0029 ± 0.000 ★	0.2474 ± 0.004 ▲	0.2373 ± 0.002 ▲
Fastjson	0.2474 ± 0.016 ▲	0.2395 ± 0.028 ▲	0.0029 ± 0.000 ★	0.4774 ± 0.181 ▲	0.4740 ± 0.181 ▲
Deeplearning4j	0.0038 ± 0.000 ▲	0.0187 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0271 ± 0.000 ▲	0.0271 ± 0.000 ▲
DSpace	0.0101 ± 0.001 ▲	0.0507 ± 0.002 ▲	0.0027 ± 0.000 ★	0.0291 ± 0.001 ▲	0.0291 ± 0.001 ▲
GSDTSR	0.0027 ± 0.000 ▲	0.0028 ± 0.000 ▲	0.0021 ± 0.000 ★	0.0253 ± 0.000 ▲	0.0253 ± 0.000 ▲
Guava	0.0335 ± 0.003 ▲	0.0405 ± 0.003 ▲	0.0028 ± 0.000 ★	0.0648 ± 0.012 ▲	0.0648 ± 0.012 ▲
IOF/ROL	0.0034 ± 0.000 ▲	0.5364 ± 0.088 ▲	0.0024 ± 0.000 ★	0.0278 ± 0.000 ▲	0.0278 ± 0.000 ▲
LexisNexis	0.7408 ± 0.114 ▲	1.5037 ± 0.290 ▲	0.0037 ± 0.000 ★	0.3745 ± 0.008 ▲	0.3710 ± 0.008 ▲
Paint Control	0.0020 ± 0.000 ★	0.0028 ± 0.000 ▲	0.0021 ± 0.000 ▲	0.0256 ± 0.000 ▲	0.0256 ± 0.000 ▲
OkHttp	0.0079 ± 0.001 ▲	0.0371 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0283 ± 0.000 ▲	0.0283 ± 0.000 ▲
Retrofit	0.005 ± 0.001 ▲	0.0178 ± 0.003 ▲	0.0026 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
ZXing	0.0156 ± 0.002 ▲	0.0229 ± 0.006 ▲	0.0026 ± 0.000 ★	0.0343 ± 0.000 ▲	0.0343 ± 0.000 ▲
			Time Budget: 80%		
Druid	0.3733 ± 0.022 ▲	0.2954 ± 0.092 ▲	0.0029 ± 0.000 ★	0.2518 ± 0.003 ▲	0.2393 ± 0.002 ▲
Fastjson	0.2587 ± 0.019 ▲	0.223 ± 0.032 ▲	0.0028 ± 0.000 ★	0.4928 ± 0.185 ▲	0.4795 ± 0.182 ▲
Deeplearning4j	0.0037 ± 0.001 ▲	0.0217 ± 0.008 ▲	0.0025 ± 0.000 ★	0.0272 ± 0.000 ▲	0.0272 ± 0.000 ▲
DSpace	0.0107 ± 0.001 ▲	0.0556 ± 0.001 ▲	0.0025 ± 0.000 ★	0.0293 ± 0.001 ▲	0.0293 ± 0.001 ▲
GSDTSR	0.0026 ± 0.000 ▲	0.0028 ± 0.000 ▲	0.0021 ± 0.000 ★	0.0253 ± 0.000 ▲	0.0253 ± 0.000 ▲
Guava	0.0349 ± 0.003 ▲	0.0405 ± 0.003 ▲	0.0028 ± 0.000 ★	0.0655 ± 0.012 ▲	0.0649 ± 0.012 ▲
IOF/ROL	0.0033 ± 0.001 ▲	0.6649 ± 0.094 ▲	0.0023 ± 0.000 ★	0.0279 ± 0.000 ▲	0.0279 ± 0.000 ▲
LexisNexis	1.0343 ± 0.451 ▲	3.8491 ± 1.681 ▲	0.0036 ± 0.000 ★	0.4130 ± 0.013 ▲	0.4090 ± 0.013 ▲
Paint Control	0.0019 ± 0.000 ★	0.0032 ± 0.001 ▲	0.0021 ± 0.000 ▲	0.0257 ± 0.000 ▲	0.0257 ± 0.000 ▲
OkHttp	0.0076 ± 0.001 ▲	0.0417 ± 0.002 ▲	0.0026 ± 0.000 ★	0.0284 ± 0.000 ▲	0.0284 ± 0.000 ▲
Retrofit	0.0052 ± 0.001 ▲	0.0215 ± 0.004 ▲	0.0025 ± 0.000 ★	0.0277 ± 0.000 ▲	0.0277 ± 0.000 ▲
ZXing	0.0188 ± 0.007 ▲	0.0254 ± 0.011 ▲	0.0026 ± 0.000 ★	0.0343 ± 0.000 ▲	0.0343 ± 0.000 ▲

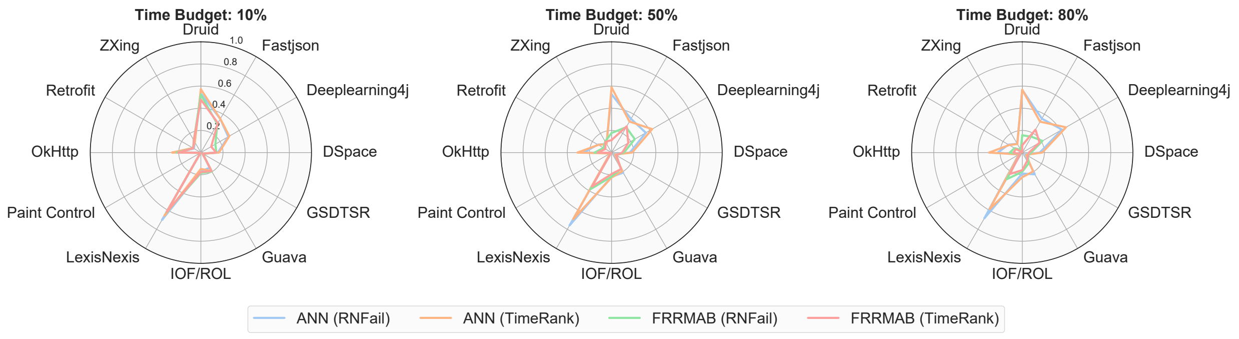


Figure 6. Radar charts - RMSE values found using NAPFD.

of failures increases with the test case volatility. This may have favored a catastrophic forgetting in the ANN.

> Finding 9. A high test case volatility combined with an increasing number of failures may be a limitation for RETECS.

On the other hand, considering the RMSE values for APFDc metric (Table 9), we observe that RETECS has better performance than COLEMAN in a restrictive scenario, in the other time budgets COLEMAN improves, been competitive in time budget of 50% and better than RETECS in time budget of 80%. In overall, COLEMAN obtained the best results using TimeRank function in 17 cases (47%), and RETECS using RNFail function in 9 cases (25%). Figure 8 shows the radar plot for RMSE values considering APFDc metric.

We observe similar charts to the ones obtained using the NAPFD metric, including a bad performance in the same systems. However, the RMSE values for APFDc are small, that is, both approaches provide good performance to reduce testing costs.

Table 8. RMSE comparison - NAPFD.

SUT	RETECS		COLEMAN	
	RNFail	TimeRank	RNFail	TimeRank
Time Budget: 10%				
Druid	0.5326 ± 0.1449 ▲	0.5716 ± 0.0749 ▲	0.5225 ± 0.0790 ▲	0.4774 ± 0.1162 ▲
Fastjson	0.3513 ± 0.0256 ▲	0.3512 ± 0.0067 ▲	0.2975 ± 0.0261 ▽	0.3080 ± 0.0333 △
Deeplearning4j	0.2858 ± 0.0818 ▽	0.2971 ± 0.0197 ▽	0.1512 ± 0.0031 ▼	0.1077 ± 0.0009 ★
DSpace	0.1529 ± 0.0014 ▼	0.1635 ± 0.0012 ▼	0.1397 ± 0.0056 ★	0.1381 ± 0.0102 ★
GSDTSR	0.0203 ± 0.0002 ★	0.0201 ± 0.0002 ★	0.0190 ± 0.0005 ★	0.0186 ± 0.0005 ★
Guava	0.1583 ± 0.0396 ▼	0.1919 ± 0.0105 ▼	0.1981 ± 0.0030 ▼	0.1930 ± 0.0022 ▼
IOF/ROL	0.1695 ± 0.0116 ▼	0.1476 ± 0.0083 ★	0.1852 ± 0.0027 ▼	0.1759 ± 0.0030 ▼
LexisNexis	0.7026 ± 0.0142 ▲	0.6583 ± 0.0599 ▲	0.6216 ± 0.0017 ▲	0.6175 ± 0.0016 ▲
Paint Control	0.0005 ± 0.0006 ★	0.0026 ± 0.0004 ★	0.0048 ± 0.0001 ★	0.0048 ± 0.0001 ★
OkHttp	0.2126 ± 0.0033 ▼	0.2613 ± 0.0117 ▽	0.2200 ± 0.0000 ▼	0.2064 ± 0.0000 ▼
Retrofit	0.0730 ± 0.0056 ★	0.0835 ± 0.0079 ★	0.0802 ± 0.0000 ★	0.0790 ± 0.0000 ★
ZXing	0.1100 ± 0.0022 ★	0.1097 ± 0.0026 ★	0.1283 ± 0.0000 ★	0.1276 ± 0.0000 ★
Time Budget: 50%				
Druid	0.5264 ± 0.1502 ▲	0.5885 ± 0.0778 ▲	0.1766 ± 0.0357 ▼	0.1129 ± 0.0327 ★
Fastjson	0.3516 ± 0.0105 ▲	0.3234 ± 0.0211 △	0.2645 ± 0.0488 ▽	0.2739 ± 0.0623 ▽
Deeplearning4j	0.3577 ± 0.0773 ▲	0.4235 ± 0.0213 ▲	0.2424 ± 0.0014 ▽	0.1743 ± 0.0006 ▼
DSpace	0.1655 ± 0.0022 ▼	0.1857 ± 0.0026 ▼	0.1116 ± 0.0292 ★	0.0959 ± 0.0271 ★
GSDTSR	0.0282 ± 0.0005 ★	0.0349 ± 0.0003 ★	0.0494 ± 0.0002 ★	0.0492 ± 0.0002 ★
Guava	0.2100 ± 0.0405 ▼	0.1951 ± 0.0114 ▼	0.1704 ± 0.0083 ▼	0.1717 ± 0.0170 ▼
IOF/ROL	0.2183 ± 0.0162 ▼	0.2258 ± 0.0129 ▼	0.2168 ± 0.0041 ▼	0.1937 ± 0.0040 ▼
LexisNexis	0.7598 ± 0.0230 ▲	0.6948 ± 0.0916 ▲	0.3820 ± 0.0074 ▲	0.3543 ± 0.0077 ▲
Paint Control	0.0071 ± 0.0005 ★	0.0160 ± 0.0010 ★	0.0049 ± 0.0004 ★	0.0048 ± 0.0003 ★
OkHttp	0.2531 ± 0.0169 ▽	0.3086 ± 0.0047 △	0.1537 ± 0.0004 ▼	0.1278 ± 0.0006 ★
Retrofit	0.1434 ± 0.0079 ★	0.1384 ± 0.0081 ★	0.0836 ± 0.0000 ★	0.0644 ± 0.0000 ★
ZXing	0.0914 ± 0.0008 ★	0.0897 ± 0.0043 ★	0.1158 ± 0.0000 ★	0.1114 ± 0.0000 ★
Time Budget: 80%				
Druid	0.5656 ± 0.1268 ▲	0.5667 ± 0.1099 ▲	0.1562 ± 0.0312 ▼	0.0666 ± 0.0182 ★
Fastjson	0.3540 ± 0.0105 ▲	0.3221 ± 0.0160 △	0.1656 ± 0.0281 ▼	0.2392 ± 0.0739 ▽
Deeplearning4j	0.4147 ± 0.0949 ▲	0.4567 ± 0.0145 ▲	0.2111 ± 0.0031 ▼	0.1805 ± 0.0013 ▼
DSpace	0.1581 ± 0.0028 ▼	0.1807 ± 0.0031 ▼	0.0803 ± 0.0189 ★	0.0609 ± 0.0271 ★
GSDTSR	0.0312 ± 0.0015 ★	0.0398 ± 0.0003 ★	0.0597 ± 0.0002 ★	0.0595 ± 0.0002 ★
Guava	0.2268 ± 0.0309 ▼	0.1968 ± 0.0190 ▼	0.1131 ± 0.0498 ★	0.0876 ± 0.0572 ★
IOF/ROL	0.1893 ± 0.0155 ▼	0.2226 ± 0.0152 ▼	0.1733 ± 0.0040 ▼	0.1585 ± 0.0033 ▼
LexisNexis	0.6857 ± 0.0433 ▲	0.5898 ± 0.1237 ▲	0.2773 ± 0.0049 ▽	0.2230 ± 0.0037 ▼
Paint Control	0.0360 ± 0.0013 ★	0.0330 ± 0.0022 ★	0.0240 ± 0.0006 ★	0.0236 ± 0.0007 ★
OkHttp	0.2219 ± 0.0260 ▼	0.3007 ± 0.0067 △	0.1131 ± 0.0004 ★	0.0795 ± 0.0007 ★
Retrofit	0.1437 ± 0.0161 ★	0.1362 ± 0.0120 ★	0.0808 ± 0.0000 ★	0.0652 ± 0.0000 ★
ZXing	0.0912 ± 0.0007 ★	0.0892 ± 0.0035 ★	0.0283 ± 0.0000 ★	0.0019 ± 0.0000 ★

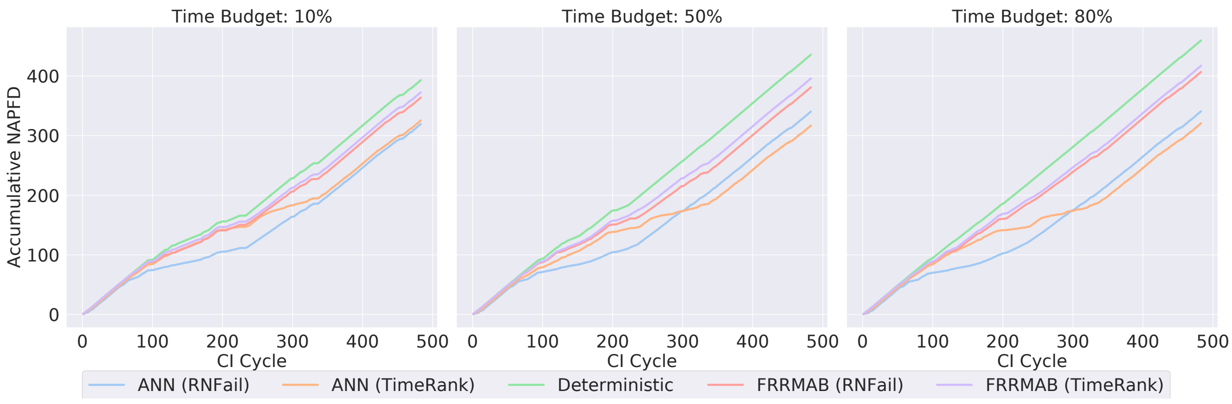


Figure 7. Accumulative NAPFD values for Deeplearning4j system.

5.6 Answering our general question

Analysing the results of our RQs in this section we aim at answering our general question by defining a scale of RMSE magnitude to represent how far the solutions found by the learning approaches are from the optimal solutions, as follows:

$$\text{RMSE Magnitude} = \begin{cases} \text{very near} & \text{if } RMSE < 0.15 \\ near & \text{if } 0.15 \leq RMSE < 0.23 \\ reasonable & \text{if } 0.23 \leq RMSE < 0.30 \\ far & \text{if } 0.30 \leq RMSE < 0.35 \\ \text{very far} & \text{if } 0.35 \leq RMSE \end{cases} \tag{7}$$

where i) the very near category ("★") represents an approximated optimal performance, that is, the prioritization generated by the approach is almost equivalent to the one generated by deterministic; ii) the near category

Table 9. RMSE comparison - APFDc.

SUT	RETECS RNFail	RETECS TimeRank	COLEMAN RNFail	COLEMAN TimeRank
Time Budget: 10%				
Druid	0.3798 ± 0.1186 ▲	0.4221 ± 0.0579 ▲	0.5072 ± 0.0922 ▲	0.4750 ± 0.1176 ▲
Fastjson	0.3340 ± 0.0244 △	0.3343 ± 0.0054 △	0.2935 ± 0.0246 ▽	0.3079 ± 0.0334 △
Deeplearning4j	0.0930 ± 0.0483 ★	0.0966 ± 0.0407 ★	0.1922 ± 0.0036 ▼	0.1839 ± 0.0014 ▼
DSpace	0.1471 ± 0.0012 ★	0.1567 ± 0.0013 ▼	0.1353 ± 0.0044 ★	0.1353 ± 0.0100 ★
GSDTSR	0.0193 ± 0.0002 ★	0.0188 ± 0.0002 ★	0.0191 ± 0.0005 ★	0.0186 ± 0.0005 ★
Guava	0.1203 ± 0.0237 ★	0.1380 ± 0.0105 ★	0.1977 ± 0.0024 ▼	0.1942 ± 0.0019 ▼
IOF/ROL	0.1696 ± 0.0116 ▼	0.1478 ± 0.0083 ★	0.1862 ± 0.0027 ▼	0.1768 ± 0.0030 ▼
LexisNexis	0.7056 ± 0.0140 ▲	0.6625 ± 0.0583 ▲	0.6284 ± 0.0017 ▲	0.6243 ± 0.0016 ▲
Paint Control	0.0005 ± 0.0007 ★	0.0033 ± 0.0006 ★	0.0060 ± 0.0001 ★	0.0060 ± 0.0001 ★
OkHttp	0.1868 ± 0.0028 ▼	0.2273 ± 0.0120 ▼	0.2083 ± 0.0000 ▼	0.1982 ± 0.0000 ▼
Retrofit	0.0636 ± 0.0058 ★	0.0744 ± 0.0084 ★	0.0816 ± 0.0000 ★	0.0803 ± 0.0000 ★
ZXing	0.0940 ± 0.0031 ★	0.0937 ± 0.0035 ★	0.1256 ± 0.0000 ★	0.1243 ± 0.0000 ★
Time Budget: 50%				
Druid	0.3365 ± 0.1361 △	0.4203 ± 0.0602 ▲	0.1650 ± 0.0437 ▼	0.1050 ± 0.0341 ★
Fastjson	0.2018 ± 0.0097 ▼	0.1956 ± 0.0446 ▼	0.2594 ± 0.0492 ▽	0.2691 ± 0.0602 ▽
Deeplearning4j	0.2061 ± 0.0591 ▼	0.1983 ± 0.0248 ▼	0.2513 ± 0.0029 ▽	0.2363 ± 0.0011 ▽
DSpace	0.1401 ± 0.0028 ★	0.1568 ± 0.0033 ▼	0.1085 ± 0.0259 ★	0.0969 ± 0.0252 ★
GSDTSR	0.0267 ± 0.0004 ★	0.0290 ± 0.0003 ★	0.0473 ± 0.0002 ★	0.0471 ± 0.0002 ★
Guava	0.1114 ± 0.0253 ★	0.1049 ± 0.0199 ★	0.1606 ± 0.0084 ▼	0.1708 ± 0.0184 ▼
IOF/ROL	0.2186 ± 0.0166 ▼	0.2315 ± 0.0122 ▽	0.2209 ± 0.0041 ▼	0.1981 ± 0.0038 ▼
LexisNexis	0.7327 ± 0.0172 ▲	0.6780 ± 0.0868 ▲	0.3893 ± 0.0073 ▲	0.3636 ± 0.0075 ▲
Paint Control	0.0059 ± 0.0006 ★	0.0315 ± 0.0019 ★	0.0126 ± 0.0003 ★	0.0126 ± 0.0003 ★
OkHttp	0.2164 ± 0.0290 ▼	0.2114 ± 0.0040 ▼	0.1360 ± 0.0004 ★	0.1192 ± 0.0005 ★
Retrofit	0.1138 ± 0.0086 ★	0.1069 ± 0.0098 ★	0.0731 ± 0.0000 ★	0.0572 ± 0.0000 ★
ZXing	0.0420 ± 0.0018 ★	0.0407 ± 0.0071 ★	0.1078 ± 0.0000 ★	0.1059 ± 0.0000 ★
Time Budget: 80%				
Druid	0.4928 ± 0.1520 ▲	0.5337 ± 0.1015 ▲	0.1562 ± 0.0396 ▼	0.0538 ± 0.0218 ★
Fastjson	0.2859 ± 0.0157 ▽	0.2739 ± 0.0318 ▽	0.1679 ± 0.0268 ▼	0.2309 ± 0.0689 ▽
Deeplearning4j	0.2728 ± 0.0838 ▽	0.2104 ± 0.0238 ▼	0.2842 ± 0.0034 ▽	0.3058 ± 0.0014 △
DSpace	0.1169 ± 0.0030 ★	0.1454 ± 0.0038 ★	0.0782 ± 0.0171 ★	0.0707 ± 0.0260 ★
GSDTSR	0.0284 ± 0.0006 ★	0.0305 ± 0.0003 ★	0.0541 ± 0.0002 ★	0.0539 ± 0.0002 ★
Guava	0.1686 ± 0.0333 ▼	0.1596 ± 0.0281 ▼	0.1134 ± 0.0524 ★	0.0922 ± 0.0584 ★
IOF/ROL	0.1870 ± 0.0160 ▼	0.2291 ± 0.0137 ▼	0.1797 ± 0.0037 ▼	0.1652 ± 0.0032 ▼
LexisNexis	0.6459 ± 0.0365 ▲	0.5812 ± 0.1008 ▲	0.2890 ± 0.0045 ▽	0.2424 ± 0.0037 ▽
Paint Control	0.0321 ± 0.0011 ★	0.0431 ± 0.0025 ★	0.0301 ± 0.0005 ★	0.0295 ± 0.0006 ★
OkHttp	0.2246 ± 0.0408 ▼	0.1846 ± 0.0074 ▼	0.1058 ± 0.0005 ★	0.0791 ± 0.0007 ★
Retrofit	0.1084 ± 0.0152 ★	0.0985 ± 0.0136 ★	0.0702 ± 0.0000 ★	0.0544 ± 0.0000 ★
ZXing	0.0431 ± 0.0014 ★	0.0437 ± 0.0116 ★	0.0258 ± 0.0000 ★	0.0030 ± 0.0000 ★

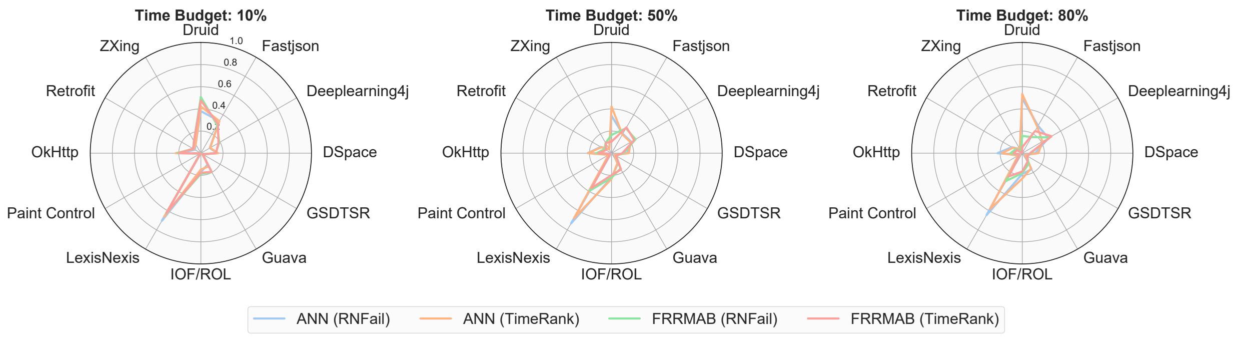


Figure 8. Radar charts - RMSE values found using APFDc.

("▼") represents reaching optimal performance, and that some improvements are required; iii) the reasonable category ("▽") represents the minimum acceptable performance. Solutions in this category are acceptable and are related to the cases in which the SUT behavior, or possibly the constraints, can make the TCP task hard. In this sense, the approach requires some refinement for improvements, such as a dedicated tuning for the SUT; iv) the far category ("△") represents unsatisfactory performance, and that meaningful improvements are required; and v) the very far category ("▲") includes solutions that are far away from to be useful and considered reasonable. By analogy and to a better visualization, we represent the RMSE magnitude with the same symbols used to represent the effect size magnitude in Tables 8 and 9. In this way, we generate Table 10 that presents the distribution of each magnitude for the NAPFD and APFDc values found by COLEMAN and RETECS.

In this path, we consider an approach that finds reasonable solutions when RMSE < 0.3. Moreover, other measures suggest such an affirmation. For instance, with 10% of the available time to execute the test cases and considering RMSE for NAPFD values, COLEMAN obtains reasonable solutions with RNFail function in 10

Table 10. RMSE magnitudes for NAPFD and APFDc values found by COLEMAN and RETECS.

Scale	NAPFD				APFDc			
	RETECS		COLEMAN		RETECS		COLEMAN	
	RNFail	TimeRank	RNFail	TimeRank	RNFail	TimeRank	RNFail	TimeRank
	Time Budget: 10%							
★ very near	4 (33%)	5 (42%)	5 (42%)	6 (50%)	7 (58%)	7 (58%)	5 (42%)	5 (42%)
▼ near	4 (33%)	2 (17%)	4 (33%)	3 (25%)	2 (17%)	2 (17%)	4 (33%)	4 (33%)
▽ reasonable	1 (8%)	2 (17%)	1 (8%)	0 (0%)	0 (0%)	0 (0%)	1 (8%)	0 (0%)
△ far	0 (0%)	0 (0%)	0 (0%)	1 (8%)	1 (8%)	1 (8%)	0 (0%)	1 (8%)
▲ very far	3 (25%)	3 (25%)	2 (17%)	2 (17%)	2 (17%)	2 (17%)	2 (17%)	2 (17%)
	Time Budget: 50%							
★ very near	4 (33%)	4 (33%)	5 (42%)	7 (58%)	6 (50%)	5 (42%)	6 (50%)	7 (58%)
▼ near	3 (25%)	3 (25%)	4 (33%)	3 (25%)	4 (33%)	4 (33%)	3 (25%)	2 (17%)
▽ reasonable	1 (8%)	0 (0%)	2 (17%)	1 (8%)	0 (0%)	1 (8%)	2 (17%)	2 (17%)
△ far	0 (0%)	2 (17%)	0 (0%)	0 (0%)	1 (8%)	0 (0%)	0 (0%)	0 (0%)
▲ very far	4 (33%)	3 (25%)	1 (8%)	1 (8%)	1 (8%)	2 (17%)	1 (8%)	1 (8%)
	Time Budget: 80%							
★ very near	4 (33%)	4 (33%)	7 (58%)	8 (67%)	5 (42%)	5 (42%)	7 (58%)	8 (67%)
▼ near	4 (33%)	3 (25%)	4 (33%)	3 (25%)	3 (25%)	4 (33%)	3 (25%)	1 (8%)
▽ reasonable	0 (0%)	0 (0%)	1 (8%)	1 (8%)	2 (17%)	1 (8%)	2 (17%)	2 (17%)
△ far	0 (0%)	2 (17%)	0 (8%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (8%)
▲ very far	4 (33%)	3 (25%)	0 (0%)	0 (0%)	2 (17%)	2 (17%)	0 (0%)	0 (0%)
Total of Reasonable Solutions[1]	25 (69%)	23 (64%)	33 (92%)	32 (89%)	29 (81%)	29 (81%)	33 (92%)	31 (86%)

[1]The reasonable solutions are that ones with RMSE < 0.3.

out of 12 cases, while with the budgets of 50% and 80% obtains, respectively 11 and 12 cases. However, on the other hand, RETECS using RNFail function obtains reasonable solutions in 9 out of 12 cases with the budget of 10%, and in 8 cases with the budgets of 50% and 80%.

Considering RMSE values for APFDc metric, COLEMAN using RNFail function, finds reasonable solutions in 10 cases, out of 12, for a time budget of 10%, and 11 and 12 cases for, respectively, the budgets of 50% and 80%. While RETECS using RNFail finds reasonable solutions in 9 out of 12 cases with the budget of 10%, and in 10 cases for the budgets of 50% and 80%. In such cases, we can conclude that the approaches have, in overall, a good performance.

Considering all 72 cases - all systems, budgets and both measures NAPFD and APDFc - COLEMAN obtained reasonable solutions in 66 cases (92%) using RNFail, and using TimeRank in 63 (88%). On the other hand, RETECS obtained reasonable solutions in 54 cases (75%) using RNFail and using TimeRank in 52 (72%). In overall, RNFail produced more reasonable solutions than TimeRank for both approaches. However, COLEMAN using TimeRank obtained the best performance obtaining the best RMSE values (highlight in gray in Tables 8 and 9) in 24 cases for NAPFD and in 17 for APFDc, followed by RETECS using RNFail that obtained 5 cases for NAPFD and 9 for APFDc.

Finding 10. Based on the proposed scale, COLEMAN finds reasonable solutions in 92% of the cases and RETECS in 75%. The less restrictive the budget the greater the number of reasonable solutions found by both approaches. We can then conclude that the solutions generated are very close to the optimal ones.

5.7 Discussions and Implications

In this section, we discuss some implications of our findings regarding the application and limitations of the approaches.

Guidelines for application. Both approaches generate reasonable solutions compared with optimal ones. COLEMAN and RETECS are able to obtain reasonable solutions in, respectively, 92% and 75% of the cases. That is, they provided solutions with a high quality and spending around one second in the worst case to execute. For this reason, both approaches are applicable in real scenarios, contributing to reducing costs by decreasing the time spent in the CI cycle. RETECS presents better performance with RNFail function. This happens with all measures evaluated. In contrast with COLEMAN, which performs better with TimeRank.

COLEMAN outperforms RETECS in the great majority of cases. It is important to highlight that this happens for the systems that can be considered hard cases, and considering all measures. However, the use of RETECS is indicated in a restrictive budget regarding early fault detection with cost consideration, in this case, test case execution. However, overall considering this cost does not seem to impact the results, nor the performance of COLEMAN, which does not include time in its formulation.

Another point to be considered in future research is to evaluate if the performance of the approaches is somehow impacted by how well a given project is adopting CI. For example, although many CI projects claim to use CI, they actually fail to implement CI practices. This failure to implement CI has been coined as CI Theather Felidre et al. (2019).

Limitations and Improvements. We observe that both approaches have some limitations to learn with few historical test data. The following characteristics are drawbacks for the learning approaches: systems with a small number of CI Cycles, with peaks of failures, and a large test case set, in which many failures are distributed over many test cases. In this way, a possible research direction is to propose a hybrid approach. An algorithm with good performance with few historical data could be used to overcome that limitation in the first commits. After with enough information RETECS or COLEMAN would be used. Another possible improvement is the use of Long Short Term Memory (LSTM) networks (Hochreiter and Schmidhuber, 1997). The LSTM is well suited to classify, process, and predict time series with time intervals of unknown duration. Gap length insensitivity gives LSTM an advantage over traditional ANNs (used by RETECS).

Benchmark. Our analysis revealed some interesting characteristics of the target systems that could be considered in the composition of a benchmark for future experiments. The IOF/ROL, LexisNexis, Deeplearning4j, and Druid systems can be considered the most challenging prioritization cases, due to the high volatility presented, as well as the number of test cases, peaks of failures, and the high number of failures distributed over many test cases. The Guava, Retrofit, and ZXing systems represent scenarios for that it is challenging to obtain expressive time reduction. The failure distribution over the test cases is low and presents small number of peaks in a few CI Cycles. In addition to this, the failing test cases vary in each CI Cycle.

6 Threats to Validity

We identified the following points that can be threats to the validity of our results.

Internal Validity: the set of parameters used for the approaches is a threat. It is possible that using an automatic configuration setting, the results can get improvements. To minimize such a threat, we used parameters from previous experiments (Prado Lima and Vergilio, 2020a; Spieker et al., 2017) reported in the literature.

External Validity: the datasets used can be considered a threat. For this, we used a relevant set of systems with different behaviors and aspects concerning the number of failures and test cases. But the results cannot be generalized.

Conclusion Validity: the measures used is a threat to the analysis conducted. Other TCP measures could lead to different results. To mitigate this threat, we chose distinct measures largely used in the TCP literature that better deal with the time budgets and allow us to analyze different perspectives.

Another threat concerns the RMSE magnitude scale. Such magnitudes were obtained based on our analysis, observations, SUTs behavior, and by making correlation with NAPFD and APFDc values. Other researchers can observe different aspects and propose a different scale.

7 Concluding Remarks

In this paper, we evaluate how far the solutions obtained by TCPCI Ranking-to-Learn approaches, RETECS and COLEMAN, are from optimal solutions produced by a deterministic approach (ground truth). We analyzed three test budgets and two reward functions: Reward Based on Failures and Reward Based on Time-Rank, concerning twelve large-scale real-world software systems. Six measures are used to evaluate: fault detection capability (and cost consideration), early fault detection, time reduction percentage in the CI cycles, prioritization time, and distance from the approximated solution.

Regarding the application of the approaches, RETECS reaches the best performance with RNFail function, in a less restrictive budget of 10%, and APFDc considering test duration as cost. COLEMAN reaches the best performance with TimeRank function and mainly for budgets of 50% and 80%. Overall, COLEMAN outperforms RETECS in the great majority of the cases, considering all systems, budgets, and measures.

Regarding our RQ, we can conclude that both approaches are applicable in real scenarios, taking a negligible time to execute and reducing the CI cycle's time cost. Considering all cases - all systems, budgets and both measures NAPFD and APDFc - COLEMAN and RETECS produce solutions that are close to the optimal ones in, respectively, 92% and 75% of the cases.

We observe that a high test case volatility, i.e., test case addition or removing along with the CI Cycles, and a high number of failures distributed over many test cases make the problem hard for both approaches. Other findings are that a few cycles can hamper the learning process and that the reduction time in a CI cycle also depends on the test case duration.

Future work includes the use of other evaluation measures to evaluate the approaches. Other systems should be used with a greater number of failures and test cases to allow scalability evaluation.

Acknowledgements

The work is supported by the Brazilian funding agencies CAPES and CNPq (Grant 305968/2018).

References

Bajaj, A. and Sangwan, O. P. (2019). A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms. IEEE Access, 7:126355–126375.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The goal question metric approach. Encyclopedia of software engineering, 2(1994):528–532.

Bertolino, A., Guerriero, A., Breno Miranda, R. P., and Russo, S. (2020). Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In 42nd International Conference on Software Engineering, ICSE'20, pages 1–12, New York, NY, USA. ACM.

Busjaeger, B. and Xie, T. (2016). Learning for Test Prioritization: An Industrial Case Study. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 975–980, New York, NY, USA. ACM.

Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2018). A Test Case Prioritization Genetic Algorithm guided by the Hypervolume Indicator. IEEE Transactions on Software Engineering.

Duvall, P., Matyas, S., and Glover, A. (2007). Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley.

Elbaum, S., Malishevsky, A., and Rothermel, G. (2001). Incorporating varying test costs and fault severities into test case prioritization. In Proceedings of the 23rd International Conference on Software Engineering, pages 329–338.

Elbaum, S., McLaughlin, A., and Penix, J. (2014). The Google Dataset of Testing Results.

Epitropakis, M., Yoo, S., Harman, M., and Burke, E. (2015). Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA, pages 234–245, New York, NY, USA. ACM.

Felidre, W., Furtado, L., Costa, D., Cartaxo, B., and Pinto, G. (2019). Continuous integration theater. In 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 1–10, Los Alamitos, CA, USA. IEEE Computer Society.

Fowler, M. (2006). Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html.

Haghighatkhah, A., Mäntylä, M., Oivo, M., and Kuvaja, P. (2018). Test prioritization in continuous integration environments. Journal of Systems and Software, 146:80–98.

Hilton, M. (2016). Understanding and improving continuous integration. In 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 1066–1067, New York, NY, USA. ACM.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8):1735–1780.

Khatibsyarbini, M., Isa, M. A., Jawawi, D. N. A., and Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology, 93:74–93.

Kruskal, W. H. and Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. Journal of the American Statistical Association, 47(260):583–621.

Kuleshov, V. and Precup, D. (2014). Algorithms for multi-armed bandit problems. Journal of Machine Learning Research, 1:1–48.

Li, K., Fialho, A., Kwong, S., and Zhang, Q. (2014). Adaptive operator selection with bandits for a multi-objective evolutionary algorithm based on decomposition. Evolutionary Computation, IEEE Transactions on, 18(1):114–130.

Li, Z., Harman, M., and Hierons, R. M. (2007). Search Algorithms for Regression Test Case Prioritization. IEEE Transactions on Software Engineering, 33(4):225–237.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics, pages 50–60.

Marijan, D. (2015). Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS'15, pages 157–162, Washington, DC, USA. IEEE Computer Society.

Marijan, D., Gotlieb, A., and Liaaen, M. (2019). A learning algorithm for optimizing continuous integration development and testing practice. Software: Practice and Experience, 49(2):192–213.

Marijan, D., Gotlieb, A., and Sen, S. (2013). Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In IEEE International Conference on Software Maintenance, pages 540–543. IEEE.

Marijan, D., Liaaen, M., Gotlieb, A., Sen, S., and Ieva, C. (2017). TITAN: Test Suite Optimization for Highly Configurable Software. In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, ICST, pages 524–531. IEEE.

Prado Lima, J. A. and Vergilio, S. R. (2020a). A multi-armed bandit approach for test case prioritization in continuous integration environments. IEEE Transactions on Software Engineering, page 12.

Prado Lima, J. A. and Vergilio, S. R. (2020b). Multi-armed bandit test case prioritization in continuous integration environments: A trade-off analysis. In Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing, pages 21–

30, New York, NY, USA. Association for Computing Machinery.

Prado Lima, J. A. and Vergilio, S. R. (2020c). Test Case Prioritization in Continuous Integration Environments: A Systematic Mapping Study. Information and Software Technology.

Prado Lima, J. A. and Vergilio, S. R. (2021). Supplementary Material - An Evaluation of Ranking-to-Learn Approaches for Test Case Prioritization in Continuous Integration. URL *https://osf.io/x96fk/?view_only=020b612cbdd84fa38d6a974743f9d823*.

Qu, X., Cohen, M. B., and Woolf, K. M. (2007). Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In IEEE International Conference on Software Maintenance, pages 255–264.

Robbins, H. (1985). Some aspects of the sequential design of experiments. In Herbert Robbins Selected Papers, pages 169–177. Springer.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (1999). Test Case Prioritization: An Empirical Study. In Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99, pages 179–188. IEEE Computer Society.

Spieker, H., Gotlieb, A., Marijan, D., and Mossige, M. (2017). Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, pages 12–22, New York, NY, USA. ACM.

Vargha, A. and Delaney, H. D. (2000). A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics, 25(2):101–132.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers.

Xiao, L., Miao, H., and Zhong, Y. (2018). Test case prioritization and selection technique in continuous integration development environments: a case study. International Journal of Engineering & Technology, 7(2.28):332–336.

Yoo, S. and Harman, M. (2012). Regression Testing Minimization, Selection and Prioritization: A Survey. Software Testing, Verification & Reliability, 22(2):67–120.

Yu, Z., Fahid, F., Menzies, T., Rothermel, G., Patrick, K., and Cherian, S. (2019). TERMINATOR: better automated UI test case prioritization. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, FSE, pages 883–894. ACM.