2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C) | 979-8-3503-5157-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/SANER-C62648.2024.00030

Leveraging Rough Sets for Enhanced Test Case Prioritization in a Continuous Integration Context

Radu Găceanu
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
radu.gaceanu@ubbcluj.ro

Arnold Szederjesi-Dragomir
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
arnold.szederjesi@ubbcluj.ro

Andreea Vescan
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
andreea.vescan@ubbcluj.ro

***Abstract*—In the rapidly evolving landscape of Continuous Integration (CI), test case execution becomes pivotal with every code modification, rendering regression testing strategies essential. Among these, Test Case Prioritization (TCP) has become a popular way to improve the efficiency and effectiveness of software testing. Recently, researchers have been mostly looking at supervised learning methods and reinforcement learning to deal with TCP in CI. However, because of the dynamic nature of these environments, it might be worth exploring unsupervised approaches that can adapt to the inherent uncertainties without labeled data. This paper proposes *RoughTCP*, a novel approach utilizing a rough sets-based agglomerative clustering algorithm, to prioritize test cases. *RoughTCP* automatically groups and ranks tests based on their intrinsic patterns and correlations (e.g., faults, tests duration, cycles count, and total runs count) without a predefined model. This improves fault detection without the need for constant supervision and provides a more comprehensive understanding of the results by incorporating rough sets. Three sets of experiments were performed, considering data from continuous integration contexts in industrial projects. Compared to recent related work, our experiments show that the *RoughTCP* approach yields better results for budgets higher than or equal to 75% on all datasets, while sometimes also outperforming all other methods on lower budgets. This underlines the potential of unsupervised methods and, in particular, the strength of *RoughTCP* in reshaping the TCP landscape in CI environments.**

***Index Terms*—Test Case Prioritization, Continuous Integration, Rough Sets, Clustering, Faults, Duration, Cycles**

I. Introduction

In today's rapidly evolving software development landscape, Continuous Integration (CI) has emerged as a dominant practice to ensure that software remains robust and reliable throughout its life cycle. CI encompasses multiple tasks, primarily revolving around consistent integration of developers' work, automated building of software, and regression testing of new software release candidates. An important challenge in CI is the scalability of the environment, particularly in regression testing. As software development progresses and codebases expand, regression test suites can grow exponentially, becoming resource-intensive. The primary objective of regression testing [1] is to ensure that changes made to the software, such as modifications or additions, do not introduce new defects or affect existing functionality. However, large volumes of tests often make it infeasible to run all test cases within the narrow time windows characteristic of CI cycles.

To address the challenges posed by expansive test suites, techniques like Test Case Prioritization (TCP) and Test Case Selection have been proposed. TCP, for instance, aims to order test cases in a manner that amplifies the likelihood of early defect detection, taking into account factors such as code coverage, frequency of code changes, and historical failure rates. Some TCP-related research investigations used advanced systems like neural networks and reinforcement learning, while others focused on improving the data they worked with.

The current proposed paper introduces a new idea: combine the rough sets theory with clustering for Test Case Prioritization in CI. When clustering is employed, test cases are grouped together, leading to a more intuitive structure than supervised learning methods. This structure not only simplifies the choice of priority test cases, but it also ensures the rapid identification and execution of the most critical ones. Moreover, unlike supervised learning, which requires a pre-trained model, clustering provides a more intuitive, training-free structure. On the other hand, rough sets help in addressing unclear or incomplete data, as well as discovering potentially important patterns in test case behaviors. This leads to a refined identification of essential test cases for each CI cycle.

The contributions of this paper are as follows:

- Proposing the innovative *RoughTCP* approach that combines rough sets with clustering in TCP for CI.
- Designing experiments considering different perspectives of the CI environment and data.
- Evaluating the proposed method against both traditional and state-of-the-art techniques, using industrial datasets to ensure its reliability and robustness.
- Discussing the advantages of the proposed rough sets-based approach over model-training methods and other crisp machine learning approaches.

The paper is structured as follows: Section II presents the TCP along with various perspectives, and Section III discusses existing TCP approaches. Section IV outlines our *RoughTCP* approach and Section V presents the results of the designed experiments, followed by the potential benefits of the

979-8-3503-5157-6/24/$31.00 ©2024 IEEE
DOI 10.1109/SANER-C62648.2024.00030

contribution. Threats to validity are discussed in Section VI. The conclusions are provided in the last section.

II. Test Case Prioritization Problem

Many companies have embraced Continuous Integration (CI) along with numerous open-source projects using CI frameworks like Travis CI and Jenkins. In case of large projects, there could be an overwhelming number of builds and test runs that require significant time and resources. Regression testing, which is part of the integration cycle, can thus be time-consuming with test sets often including thousands of cases that may take several hours or even days to complete. Hence, in the context of CI, rerunning all test cases is not feasible. Therefore, cost-effective Regression Testing (RT) is vital to ensure that recent changes have not negatively affected previously tested functionality and to provide quick feedback on software failures. Traditional RT techniques are not always suitable due to constraints like time limitations and frequent code changes. They often focus on code analysis, which can be slow and may become outdated quickly due to frequent changes. Test Case Prioritization (TCP) techniques, which reorder test suites based on objectives like early fault detection, are more suitable when time is limited. These techniques consider the entire test suite, lowering the risk of reducing code coverage. While the ideal case is to maximize early fault detection, such information is not available until after testing. TCP techniques that rely on past failures can address this challenge.

According to [2], the Test Case Prioritization (TCP) problem can be formally defined as follows:

Definition 1: **Test Case Prioritization [2]:** a test suite, T, the set of permutations of T, PT; a function from PT to real numbers, f. The goal is to find T' ∈ PT such that: $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

From Definition 1, the function f assigns a real value to a permutation of T according to the test adequacy of the particular permutation.

Unfortunately, the current problem definition does not account for a time constraint on executing the test suite. A more comprehensive definition, termed Time-limited Test Case Prioritization, enhances the traditional Test Case Prioritization problem by adding a time constraint. This inclusion suggests that due to time constraints, it might not always be feasible to run every test case. While time is a significant factor, there could be other elements that limit the selection of test cases. Nonetheless, the following formulation can be universally adjusted without any loss of generality.

Definition 2: **Time-limited Test Case Prioritization Problem (TTCP) [3]**: find T ∈ PT such that: $\big(f(T) \geq f(T')\big) \wedge \big(\sum_{t_k \in T'} t_k.duration \leq M\big) \wedge \big(\sum_{t_k \in T} t_k.duration \leq M\big), \forall T' \in PT$, where: $t_k.duration$ represents the duration of a test case k from a test suite and M is the maximum time available for the test suite execution.

Because links between code modifications and test cases are presumed to be unavailable in this paper, historical information about test case execution must be examined. This is why, in our approach, we utilize the following definition for Adaptive Test Case Selection Problem (ATCS).

Definition 3: **Adaptive Test Case Selection Problem (ATCS) [3]**: let $T_1, T_2, \ldots T_{k-1}$ be a sequence of previously executed test suites, find T_k such that $f(T_k)$ is maximized and $\sum_{t_k \in T} t_k.duration \leq M$.

III. Related work

The selection and prioritization of test cases is a subject that has received significant attention in academic research.

Recent literature has highlighted techniques aimed at early fault detection and efficiently managing the time constraints of running an exhaustive test suite [4]–[7]. However, the unique characteristics of CI, such as its dynamic environment, where code changes are frequent and test cases are constantly added or removed, remain under-addressed. Moreover, to provide a comprehensive test, there is a need to strike a balance between ensuring a diverse test suite and focusing on new or historically error-prone test cases known for their high fault-detection capabilities. Solely prioritizing error-prone cases without sufficient diversity could lead to some tests being perpetually sidelined, and hence potentially missing new issues. To this end, various innovative methods have been proposed to enhance test case prioritization. Elbaum [8] introduces integration of regression test selection with test case prioritization, using time windows to track recent failures, an approach that can be applied to individual test suites as their execution is requested. RETECS [3] fuses reinforcement learning and neural networks, with the aim of real-time adaptability in CI cycles. By utilizing a neural network, the approach takes into account both the chosen test cases and their execution sequence. It tends to prioritize test cases that have previously identified faults during CI cycles. This ensures that the testing process finds faults as early as possible. COLEMAN [9] leverages the Multi-Armed Bandit (MAB) approach to make sequential decisions tailored to evolving environments. As opposed to reinforcement learning, MAB does not need contextual data, and its decisions only influence the reward without altering the environment's state. LeaRnTec [10] represents a novel test case prioritization strategy for CI, combining principles from information retrieval and reinforcement learning. This approach uses historical data to develop an adaptive ranking model, which not only refines its efficacy with every test case execution, but also adjusts to changes such as the addition or removal of test cases. Unlike traditional methods, this strategy avoids intensive computations, relying on test case execution history and a feedback reward mechanism. Although the most recent works advocate online strategies for TCP for CI, specifically using reinforcement learning, the authors of NEUTRON [11] consider a different approach leading to better performance over current approaches in some cases. They argue that despite these methods bypassing a training phase, they still require extensive data, questioning the real advantage of no training phase.

IV. RoughTCP: Our approach on TCP in CI based on rough sets clustering

This section presents our approach to TCP in CI settings using an agglomerative clustering algorithm based on the theory of rough sets.

A. RoughTCP overview approach

This section presents an overview of our methodology for test case prioritization. Figure 1 shows an overview of the whole process, together with the three main stages: (1) test case preprocessing, (2) test case clustering, and (3) test case prioritization.

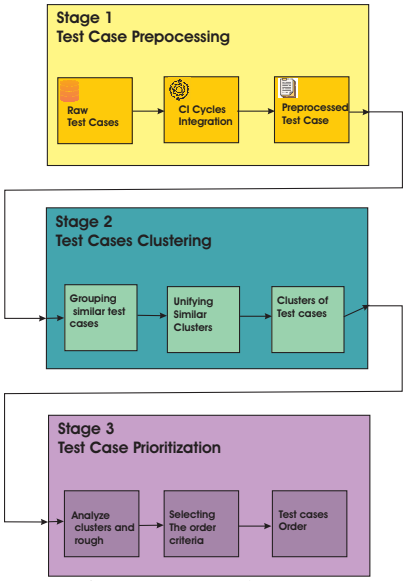

Fig. 1: Approach stages

Stage 1. In the first stage, *Test Case Preprocessing*, three industrial projects are considered. Each of these datasets offers historical data on test case executions, including outcomes (either pass or fail) across more than 300 CI cycles. To center the data around individual test cases rather than cycles or executions, the information from the three projects was restructured [11]. In this reorganization the focus was on specific properties that may be more impactful, like *duration* (execution time for the given testcase), *fault rate* (represents the number of total faults over the number of total runs), *cycles count* (the number of cycles in which the test case was executed), and *total runs count* (sum of the runs of the given test case over all cycles), thus removing redundant or irrelevant information and also reducing the size of the datasets significantly. In addition, two additional datasets were derived by augmenting them with:

- other three features regarding the average of faults at every approximately 100 cycles;
- other 11 features regarding the average of faults every approximately 30 cycles.

Stage 2. Consequently, in the next stage, *Test Case Clustering*, several experiments with multiple projects are going to be considered. A more detailed overview of the proposed methodology is shown in Figure 2 and the following sections will present thorough information regarding the datasets and the entire preprocessing stage.

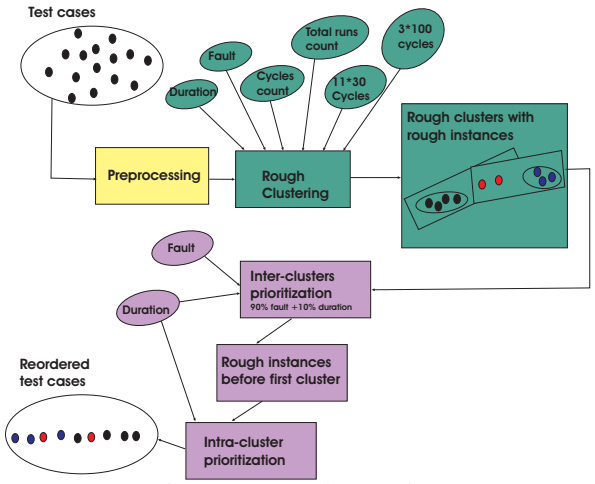

Fig. 2: Approach overview

In the proposed agglomerative clustering approach, each test case is initially assigned to its own distinct cluster. As the algorithm progresses through its iterative stages, for every test case, other similar test cases are identified. When a test case finds another one that matches its criteria, it joins the cluster of the identified test case. This iterative process allows for dynamic formation and refinement of clusters. In addition, the algorithm incorporates steps to unify clusters that are closely related or have significant overlaps, ensuring a more cohesive clustering outcome. An optional phase in the algorithm addresses possible outliers, ensuring that they are assigned to the most appropriate cluster. The algorithm may produce rough instances, i.e., test cases for which the cluster membership can not be decided with certainty, since they are similar to more than one cluster.

Stage 3. In the third stage, test case prioritization, the clusters determined by the clustering algorithm are used to obtain the ordered test cases by applying the following steps:

1) **Ordering Clusters**: the clusters are ordered based on their fault rate and duration. In this ordering, the duration is given a considerably smaller weight than the fault rate.
2) **Deprioritizing High Duration Clusters**: we perform an additional step to specifically deprioritize clusters that have a significantly high duration. Specifically, clusters with a duration 10 times greater than their neighboring clusters are moved down in order.
3) **Handling Rough Instances**: rough instances, which are potentially part of multiple clusters, are placed before the first cluster to which they belong, maintaining the order established in the previous steps.
4) **Ordering Within Clusters**: within each cluster, the test case instances are ordered based on their duration.

B. Rough sets clustering algorithm for TCP in CI

Rough sets [12] offer a robust and versatile method for dealing with uncertainty and vagueness in data and they could be applied in various domains, including medicine, engineering, finance and artificial intelligence, to handle imprecise

information and derive actionable insights. Unlike some other methods for dealing with uncertainty, like fuzzy set theory, rough sets do not require any additional information (such as membership functions in fuzzy sets). They work directly with the given data.

Given a dataset (or universe of discourse U) and an equivalence relation on this dataset, rough sets theory allows us to approximate a vague or uncertain subset $X \subseteq U$ using two precise (or "crisp") sets: the lower and upper approximations. The **lower approximation** contains all objects that definitely belong to the subset and it is defined as $R^{\downarrow}(X) = \{x \in U : [x]_R \subseteq X\}$ where $[x]_R$ denotes the equivalence class of x under R. The **upper approximation** contains all objects that possibly belong to the subset and it is defined as $R^{\uparrow}(X) = \{x \in U : [x]_R \cap X \neq \emptyset\}$. The difference between the upper and lower approximations defines the **boundary region** which is defined as $Bnd_R(X) = R^{\uparrow}(X) - R^{\downarrow}(X)$. Objects in the boundary region cannot be definitively classified as either belonging or not belonging to the subset. Then the **rough set** of X with respect to R is represented as $RS(X) = \{R^{\downarrow}(X), R^{\uparrow}(X)\}$.

Rough sets clustering [13] is an approach that uses the principles of rough sets to partition a dataset into clusters, considering the uncertainty and vagueness inherent in the data.

Definition 4: Given a dataset U (universe of discourse) and an equivalence relation R on U, the goal of **rough sets clustering** is to partition U into a set of clusters $\{C_1, C_2, \ldots, C_k\}$ such that:

- $U = \bigcup_{i=1}^{k} C_i$ and $C_i \cap C_j = \emptyset$ for $i \neq j$.
- Each cluster C_i is represented by its lower and upper approximations ($R^{\downarrow}(C_i)$, $R^{\uparrow}(C_i)$) with respect to R:
- The boundary region for each cluster C_i is given by $Bnd_R(C_i) = R^{\uparrow}(C_i) - R^{\downarrow}(C_i)$.

In the context of clustering:

- objects in the lower approximation of a cluster *definitively* belong to that cluster
- objects in the upper approximation *might* belong to the cluster
- objects in the boundary region of a cluster may belong to the boundary regions of other clusters.

In Algorithm 1 we show an overview of the rough sets clustering process from [13] adapted to the problem of TCP in CI.

The algorithm receives as input the dataset X, a maximum number of trials $imax$, and a similarity limit λ. Each data point in the dataset X, i.e., each test case is represented by an agent. The set of all agents is denoted as AG. Initially, each agent is assigned to its own unique cluster. This means that if there are n test cases (and thus n agents), there will be n clusters at the start. The algorithm performs a series of trials to refine the clustering. This is done for a maximum of $imax$ times. In each trial, every agent ($agent_k$) searches for another agent that is similar to itself based on the similarity threshold λ. If $agent_k$ finds another agent sa_k that is similar, then $agent_k$ will move to the cluster of sa_k. This process allows agents that are close to each other or similar to each other to be grouped together into the same cluster. For each cluster, a representative is chosen. The algorithm then checks if there are clusters that are similar to each other based on their representatives. If two or more clusters are found to be similar, they are unified (merged) into a single cluster. This step ensures that clusters that are close to each other or have significant overlap can be combined to form a more cohesive and meaningful cluster. It is possible for a representative to be similar to more than one other cluster representative in which case the corresponding data (rough instances) will be treated as it would belong to several clusters. Some test cases (or agents) might not fit well into any cluster. These are considered outliers. The algorithm has an optional step to handle these outliers by assigning them to the closest cluster. This step ensures that all data points are assigned to a cluster, even if they do not fit perfectly into any specific cluster. The source code of the algorithm is available at this figshare link [14].

Algorithm 1 Rough Sets Clustering

Require: X (dataset), $imax$ (number of trials), λ (similarity limit)

```
Initialize AG (set of agents) with one agent for each instance in X.
For each agent in AG, assign it to a unique cluster.
for i = 1 to imax do
    for each agent_k in AG do
        Find a similar agent (sa_k) using a similarity threshold λ.
        if sa_k is found then
            Move agent_k to the cluster of sa_k.
        end if
    end for
end for
for each cluster representative R_k do
    Find similar clusters based on a rough similarity limit.
    Update and unify clusters based on similarity.
end for
(Optional) Handle outliers by assigning them to the closest cluster.
```

C. Dataset

The datasets used in our research investigation are three industrial projects: two from ABB Robotics Norway [1] (Paint Control and IOF/ROL, for testing complex industrial robots) and Google Shared Dataset of Test Suite Results (GSDTSR) [2] are used in this study. All three projects contain information about historical test case executions, along with the verdicts (pass, fail), with CI cycles over 300. The two ABB datasets are split into daily intervals, whereas GSDTSR is split into hourly intervals as it originally provides log data of 16 days. However, the average test suite size per CI cycle in GSDTSR exceeds that in the ABB datasets. An overview of the dataset is presented in Table I.

[1] https://new.abb.com/products/robotics

[2] https://bitbucket.org/HelgeS/atcs-data/src

TABLE I: Datasets Information Overview

Project name	Test cases information			
	No. of Test Cases	*CI cycles*	*Verdicts*	*Failed*
Paint Control	89	352	25,594	19.36%
IOF/ROL	1941	320	30,319	28.43%
GSDTSR	5,555	336	1,260,617	0.25%

The authors in [11] restructured the datasets and built several matrices focusing on the information on each test case and not on the execution of the cycle. We recall here that for each test case the following information is available: test case id, duration, fault rate, cycles count, total runs count, for each cycle the number of faults over the number of runs in that cycle. The new format is available at this link [15]. The features related to each test case that are used in our research are: duration, fault rate, number of runs in all cycles, number of total executions, and a rate between the number of fails of the test case in a cycle over the number of runs in that cycle. Thus, the dataset was augmented with additional information based on the existing cycles: a new set of three features per test case was constructed considering information at every 100 cycles, and another set of eleven additional features was built based on information at every 30 cycles for each test case.

D. Design of experiments

The research investigation consists of various sets of experiments as described in Figure 3. Four features, namely, *Duration*, *Fault Rate*, *Cycles count*, and *Total runs count* are considered in all experiments. In the second set of experiments, data on test cases from every approximately 100 cycles was considered [11]. Following the experimental protocol considered in [3] (where data from every 30 CI cycles were used, considering 50% budget), we also considered this configuration in our third set of experiments.

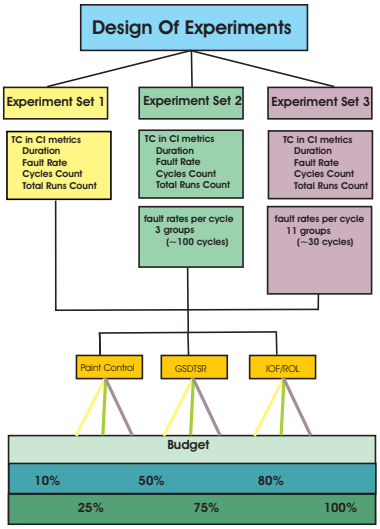

Fig. 3: Design of experiments

The projects used in the experiments are the three projects mentioned above in Section IV-C, namely, *Paint Control*, *IOF/ROL* and *GSDTSR*. For all three sets of experiments, we performed various budget percentages, 10%, 50%, and 80% as in previous studies, together with other new percentages 25%, 75%, and 100%.

E. Metrics for Analysis

The APFD (Average Percentage of Faults Detected) metric [16] is defined as follows (n is the number of test cases, m is the number of faults, and TF_f represents the position of the test case in the prioritized test suite that detects the fault f): $APFD = 1 - \frac{TF_1+TF_2+....TF_m}{n\times m} + \frac{1}{2\times n}$. The higher the value of APFD, the better fault detection.

An extension of APFD to incorporate the fact that not all test cases are executed and failures can be undetected is the Normalized APFD (NAPFD) [17]: $NAPFD = p - \frac{TF_1+TF_2+....TF_m}{n\times m} + \frac{p}{2\times n}$, where p = the number of faults detected by the prioritized test suite divided by the number of faults detected in the entire test suite. When p = 1, that is, all faults are detected, the NAPFD is the same as the APFD.

V. Results

This section contains the results of the experiments performed, first presenting an overview of the results, emphasizing comparisons from different perspectives, then outlining the result by project, and finalizing with general results extracted from the previous analysis.

A. Results overview

The results of the experiments performed using rough clustering are provided in Table II, together with the results of other approaches from Elbaum's approach [8], to RETECS [3], COLEMAN [18], and NEUTRON [11].

Our investigations employing the experiments based on the Rough Clustering (RC) are: RC-Reduced (using the 4 features discussed above), RC-3*100-cycles (using additional 3 features based on the three 100 cycles), and RC-11*30-cycles (using additional eleven features based on the eleven 30 cycles).

As stated in Section IV-D, we have performed experiments on three different datasets: Paint Control, GSDTSR, IOF/ROL. Firstly, when comparing our methods on Paint Control with other prior approaches, it becomes evident that none of them demonstrate satisfactory performance on the 10% budget. However, as the budget increases, our results exhibit substantial improvement. Notably, starting with a 75% budget, our methods outperform all other state-of-the-art approaches and continue to do so even at a 100% budget. When comparing our methods between them, we can observe that the RC-Reduced method does the best overall, but the difference is pretty small, except in the case of 10% and 80% budgets.

Additionally, while examining the GSDTSR dataset, it is evident that our methods exhibit behavior comparable to other approaches. Specifically, when considering lower budgets such as 10% or 50%, our methods perform somewhat inferior to the state-of-the-art techniques. However, when budgets are 75%, 80%, or 100%, our methods outperform all other approaches. In this case, results between our approaches are also very consistent, and all of them seem to perform at their highest potential.

Finally, our best results are observed in the context of the IOF/ROL project. Although the main idea is similar to those of

TABLE II: Experiments

Project	Budget	NAPFD								
		Elbaum	RETECS RNFail	RETECS TimeRank	COLEMAN RNFail	COLEMAN TimeRank	NEUTRON	RC-Reduced	RC-3x100-cycles	RC-11x30-cycles
Paint Control	10		**0.9078**	**0.9077**	**0.9076**	**0.9076**		0.1605	0.0393	0.0421
	25						0.2512	**0.5104**	0.4877	0.4767
	50	**0.9145**	**0.915**	0.9138	**0.915**	**0.915**	0.6004	0.7579	0.7352	0.7354
	75						0.9377	**0.9490**	0.9264	0.9265
	80		0.9162	0.9160	0.9171	0.9171		**0.9827**	0.9264	0.9265
	100						0.9940	0.9940	**0.9946**	**0.9947**
GSDTSR	10		**0.9893**	**0.9893**	**0.9894**	**0.9894**		0.8904	0.8904	0.8969
	25						0.4868	**0.9533**	**0.9533**	**0.9551**
	50	0.9891	**0.9911**	0.9906	0.9893	0.9894	0.9175	0.9880	0.9880	0.9880
	75						0.9870	**0.9967**	**0.9967**	**0.9965**
	80		0.9921	0.9914	0.9893	0.9894		**0.9977**	**0.9976**	**0.9974**
	100						0.9983	**0.9992**	**0.9992**	**0.9990**
IOF/ROL	10		**0.3704**	**0.3779**	0.3632	0.3670		0.1578	0.1004	0.1939
	25							0.4330	0.3802	**0.4718**
	50	0.4892	0.5101	0.5025	0.5046	0.5189		**0.7909**	0.7762	**0.8125**
	75							0.8784	**0.9019**	**0.8964**
	80		0.5495	0.5287	0.5569	0.5678		0.8874	**0.9107**	**0.9026**
	100							0.9180	**0.9318**	0.9199

the other projects, our methods in this particular project exhibit an earlier peak and provide superior performance across all budgets, with the exception of the 10% budget. Furthermore, there appears to be a greater difference between our respective methods, those using a greater number of characteristics, such as RC-3x100-cycles or RC-11x30-cycles, demonstrating superior performance. This is the project with the most faults, so having more granular details between instances helps the results get better.

In conclusion, RETECS and COLEMAN appear to always perform better in the 10% budget, sometimes also in the 50% budget. However, when considering the Paint Control scenario, RC-Reduced emerges as the most effective method overall. In the case of GSDTSR, any of our methods may be considered viable options. Additionally, when dealing with IOF/ROL, having a greater number of features proves advantageous, making RC-11x30-cycles the optimal choice.

It is important to mention that we have also checked the information provided in the related work, as specified at the beginning of this section, however, for LeanRnTeC [19] the data were not available for verification (the link did not work and even when we contacted the authors, until the submission date we did not obtain the available link). Moreover, the values of the LeanRnTeC [19] are the same as in COLEMAN [18], thus not influencing our results.

B. Result analysis by project

1) Results for the IOF/ROL project: We outline and detail in what follows the results obtained for the rough clustering in the case of the *IOF/ROL* project.

Figure 4 contains the results using the *IOF/ROL* project under the reduced feature set context, however, only a subset of instances were included, namely those that have a significant value with respect to duration and/or fault rate, and also with respect to cycle count and total runs count.

We also normalized the values to plot them using a spider graph (Figure 4). It may be seen that there are rough instances with high duration but low fault rate, for example, instances

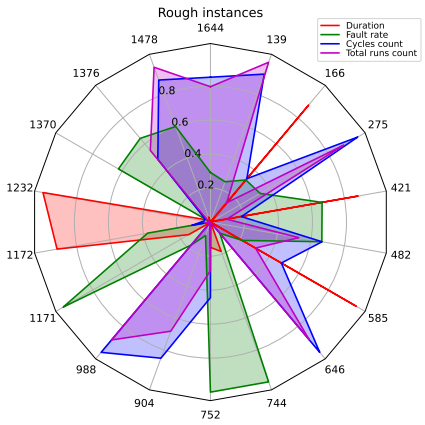

Fig. 4: Rough instances for the IOF/ROL project (reduced approach)

585 or 1172, but with different cycle counts. There are also instances with high values for both duration and fault rate; however, they have low values for cycle count and total runs count. The aforementioned plot reveals several instances that are important to be incorporated and executed in subsequent cycles. Another interesting situation involves instances 275 and 1478 which exhibit higher cycle counts and runs counts, yet display a low fault rate. This is an indication of an important feature which should be tested at each build to ensure correct functionality. Other instances may be observed having different behavior, as per example instance 752 with high fault rate but low cycle count and runs count, meaning a test case discovering a fault but not executed many times, and being maybe a functionality implemented in a specific sprint.

Analyzing the results from the perspective of rough clusters, other insights are outlined. Figure 5 contains the instances in the rough clusters that have special values among the instances in that specific cluster. The data are normalized and then plotted onto the spider charts.

For cluster 106, in Figure 5 it can be observed that there are three subclasses: one based on instances 764 and 469 with

a high number of total runs and a medium fault rate, another one still with a high number of total runs but with a lower fault rate, and the last one with high duration, low number of runs and low fault rate. Thus, the test cases in this cluster were run many times with a range of fault rate from low to high, but also test cases with high duration that were run only a few times and with a low fault rate.

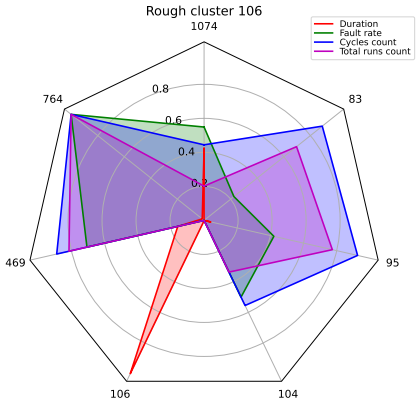

Fig. 5: Rough cluster 106 for the IOF/ROL project (reduced approach)

2) Results for the Paint Control project: We outline and detail in what follows the results obtained for the rough clustering in the case of the *Paint Control* project.

The results information for the *Paint Control* project, namely, the number of rough clusters along with the cluster and the number of rough instances with the rough instances for each experiment carried out are provided next: RC-Reduced has found 4 rough clusters (15, 20, 52, 55) and 0 rough instances, RC-3*100-cycles has found 10 rough clusters (23, 24, 39, 46, 52, 54, 55, 56,78, 87) and 1 rough instance (15), and RC-11*30-cycles has found 10 rough clusters (15, 23, 24, 39, 52, 54, 55, 60, 71, 80, 87) and no rough instances. Investigating the results, it may be seen that rough instances were found only in the experiment with 100 cycles. Specifically, a single instance, labeled as instance 15, exhibited a fault rate of 0.21 and was executed over a span of 267 cycles. This particular test case might be significant as it probably verifies a fundamental and crucial feature. Furthermore, the relatively low fault rate suggests that the functionality may have been initially flawed during the development phase but subsequently rectified.

3) Results for the GSDTSR project: We outline and detail in what follows the results obtained in the case of the *GSDTSR* project.

The results information for the *GSDTSR* project, namely, the number of rough clusters along with the cluster and the number of rough instances, are provided next. RC-Reduced has found 9 rough clusters (45, 735, 963, 1029, 1475, 1731, 2810, 3861, 5198), RC-3*100-cycles has found 10 rough clusters (735, 963, 1004, 1029, 1731, 2619, 2810, 3332, 3618, 4866), and RC-11*30-cycles has found 5 clusters (1004, 1029, 1731, 2810, 3332). No rough instances were found; however, by analyzing the rough clusters, we may outline that some of the instances are common in all three rough clustering methods having in common instances with similar durations but different cycles. Some other clusters are only common in two of the rough clustering methods with large number of runs and also some with also big durations. Other clusters are only common to one particular rough clustering method, for example in the Reduced context, the clusters have similar cycles count but may be split in two larger groups considering duration.

C. Summary of results

In what follows, we are summarizing the obtained results and the analysis.

Overall, rough clustering obtains better results in about 50% of the cases (for budgets 10%, 50%, and 80%). When considering also the other budgets (25%, 75%, and 100%) better results are obtained in 86.66% as seen in Table II. At the budget of 80%, rough clustering is better than the other existing methods in the research literature. Out of the 4 cases where rough clustering is better, 2 are for the RC-Reduced approach, 1 for RC-3*100-cycles approach, and 1 for RC-11*30-cycles approach.

Comparing only rough clustering methods, the order of best results is 7 best for the RC-11*30-cycles approach, 6 for RC-Reduced, and 5 for RC-3*100-cycles approach.

Comparing only rough clustering methods (see Figure 6), the order of best results is 7 best for the RC-11*30-cycles approach, 6 for RC-Reduced, and 5 for RC-3*100-cycles approach.

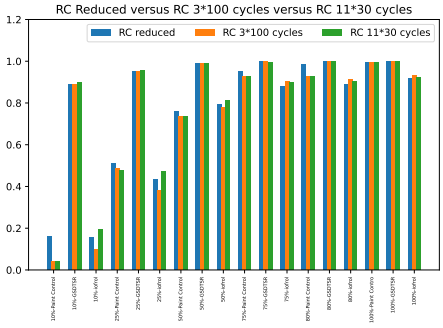

Fig. 6: RC Reduced versus RC 3*100 cycles versus RC 11*30 cycles per project and per budget

When comparing based on the projects, the best results for low budgets (10, 25, 50) are obtained for the *GSDTSR* and *IOF/ROL* projects with the RC-11*30-cycles approach and for the *Paint Control* project with RC-Reduced, but for higher budgets (80, 100) better results are still obtained for the *GSDTSR* and *IOF/ROL* projects for the RC-Reduced and RC-3*100-cycles approaches.

D. Potential benefits of the contributions

Implications for researchers and practitioners on the contributions of the paper are outlined in this section.

For the research community, our results provide new insight into other perspectives when addressing the TCP problem by

proposing a new method that focuses only on existing data on each test case based on previous execution cycles and does not need labeling (as supervised learning methods do). In addition, we have outlined potential directions for future investigations, including incorporating requirements for traceability.

For the practitioner community, our approach can be incorporated into their regression testing process, allowing them to automatically run test cases that are more likely to find errors and to find them as soon as possible using a plugin in the IDE environment. Another implication refers to pinpointing practitioners to focus their attention on novel aspects regarding regression testing and not staying stuck with the previous approaches, for example, incorporating links between requirements and test cases.

VI. Threats to validity

Internal. The first threat to internal validity is the influence of random decisions on the results, and to address the potential risk, we repeated our experiments 30 times, reporting average.

Construct Validity. The datasets contain few features related to the test cases. Although we think that a subset of features could be enough for a proper TCP, we tried multiple variants of the same datasets with different features to possibly counteract the bias created by using only a few features.

External Validity. Our technique is evaluated on three datasets obtained from industry sources. A greater number of datasets should have been utilized in our study, but to the best of our knowledge, there are no other datasets that encompass the necessary information, particularly with respect to the historical execution of the test cases.

VII. Conclusion

In regression testing, Test Case Prioritization is one of the strategies to be used, which aims to order the test cases based on different, well-defined criteria. The proposed *RoughTCP* approach uses a rough set clustering algorithm to group and rank test cases based on their intrinsic patterns in information about faults, duration, and continuous integration cycles. The approach was rigorously validated using data from three distinct industrial projects, each providing information from their continuous integration environments, such as duration, faults, or information from up to 350 cycles.

Acknowledgment

This work was funded by the Ministry of Research, Innovation, and Digitization, CNCS/CCCDI - UEFISCDI, project number PN-III-P1-1.1-TE2021-0892 within PNCDI III.

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge University Press, 2016.

[2] T. L. Graves, M. J. Harrold, J. Kim, A. Porters, and G. Rothermel, "An empirical study of regression test selection techniques," in *Proceedings of the 20th International Conference on Software Engineering*, 1998, pp. 188–197.

[3] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "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*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 12–22. [Online]. Available: https://doi.org/10.1145/3092703.3092709

[4] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja, "Test prioritization in continuous integration environments," *Journal of Systems and Software*, vol. 146, pp. 80–98, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218301730

[5] D. Marijan, A. Gotlieb, M. Liaaen, S. Sen, and C. Ieva, "Titan: Test suite optimization for highly configurable software," 2017.

[6] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "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*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 12–22. [Online]. Available: https://doi.org/10.1145/3092703.3092709

[7] J. A. Prado Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Information and Software Technology*, vol. 121, p. 106268, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300185

[8] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 235–245. [Online]. Available: https://doi.org/10.1145/2635868.2635910

[9] J. A. P. Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 453–465, 2022.

[10] S. Omri and C. Sinz, "Learning to rank for test case prioritization," in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*, 2022, pp. 16–24.

[11] A. Vescan, R. Găceanu, and A. Szederjesi-Dragomir, "Neural network-based test case prioritization in continuous integration," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2023, pp. 68–77.

[12] Z. Pawlak, "Rough sets," *International journal of computer & information sciences*, vol. 11, pp. 341–356, 1982.

[13] R. D. Găceanu, A. Szederjesi-Dragomir, H. F. Pop, and C. Sârbu, "Abarc: An agent-based rough sets clustering algorithm," *Intelligent Systems with Applications*, vol. 16, p. 200117, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667305322000552

[14] anonymous review, "Leveraging rough sets for enhanced test case prioritization in a continuous integration context," accessed December 2023. [Online]. Available: https://doi.org/10.6084/m9.figshare.24852729.v1

[15] A. Vescan, R. Gaceanu, and A. Szederjesi-Dragomir, "Neutron (neural network based test case prioritization in continuous integration)," accessed July 2023. [Online]. Available: https://figshare.com/s/8721c5c6609396937d56

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

[17] X. Qu, M. Cohen, and K. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *2007 IEEE International Conference on Software Maintenance*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2007. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSM.2007.4362638

[18] J. A. P. Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 453–465, 2022.

[19] S. Omri and C. Sinz, "Learning to rank for test case prioritization," in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*, 2022, pp. 16–24.