The Journal of Systems and Software 153 (2019) 86–104

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Employing rule mining and multi-objective search for dynamic test case prioritization

Dipesh Pradhan[a,b,*], Shuai Wang[c], Shaukat Ali[a], Tao Yue[a], Marius Liaaen[d]

[a] Simula Research Laboratory, Oslo, Norway
[b] University of Oslo, Oslo, Norway
[c] Testify AS, Oslo, Norway
[d] Cisco Systems, Oslo, Norway

ARTICLE INFO

Article history:
Received 15 May 2018
Revised 8 January 2019
Accepted 28 March 2019
Available online 29 March 2019

Keywords:
Multi-objective optimization
Rule mining
Dynamic test case prioritization
Search
Black-box regression testing

ABSTRACT

Test case prioritization (TP) is widely used in regression testing for optimal reordering of test cases to achieve specific criteria (e.g., higher fault detection capability) as early as possible. In our earlier work, we proposed an approach for black-box dynamic TP using rule mining and multi-objective search (named as REMAP) by defining two objectives (fault detection capability and test case reliance score) and considering test case execution results at runtime. In this paper, we conduct an extensive empirical evaluation of REMAP by employing three different rule mining algorithms and three different multi-objective search algorithms, and we also evaluate REMAP with one additional objective (estimated execution time) for a total of 18 different configurations (i.e., 3 *rule mining algorithms* × 3 *search algorithms* × 2 *different set of objectives*) of REMAP. Specifically, we empirically evaluated the 18 variants of REMAP with 1) two variants of random search while using two objectives and three objectives, 2) three variants of greedy algorithm based on one objective, two objectives, and three objectives, 3) 18 variants of static search-based prioritization approaches, and 4) six variants of rule-based prioritization approaches using two industrial and three open source case studies. Results showed that the two best variants of REMAP with two objectives and three objectives significantly outperformed the best variants of competing approaches by 84.4% and 88.9%, and managed to achieve on average 14.2% and 18.8% higher Average Percentage of Faults Detected per Cost ($APFD_c$) scores.

1. Introduction

Modern software is developed at a rapid pace to add new features or fix detected bugs continuously. This can lead to new faults into the previously tested software and to ensure that no new bugs are introduced, regression testing is frequently applied in the industry (Onoma et al., 1998; Borjesson and Feldt, 2012; Rothermel and Harrold, 1997). Specifically, in regression testing previously developed test cases are used to validate software changes. However, regression testing is an expensive maintenance process that can consume up to 80% of the overall testing budgets (Kaner, 1997; Chittimalli and Harrold, 2009), and it might not be possible to execute all the test cases when the testing resources (e.g., execution time) are limited.

Test case prioritization (TP) is one of the most widely used approaches to improve regression testing to schedule test cases for achieving certain criteria (e.g., code coverage) as quickly as possible (Rothermel et al., 2001, 1999; Li et al., 2007; Elbaum et al., 2014a). Most of the existing techniques for TP aim to find the faults as soon as possible, however, an ability of a test case to detect a fault is determined only after executing it. The execution results of the executed test cases at runtime are not usually used to prioritize the test cases dynamically by the existing TP techniques (Rothermel et al., 1999; Elbaum et al., 2002; Wong et al., 1997; Kim and Porter, 2002; Khalilian et al., 2012; Noor and Hemmati, 2015), i.e., they produce a list of static prioritized test cases. Based on our collaboration with Cisco Systems (Wang et al., 2013, 2014) focusing on cost-effectively testing video conferencing systems (VCSs), we noticed that there might exist underlying relations among the executions of test cases, which test engineers are not aware of when developing these test cases. For example, when the test case (T_1) that tests the amount of free memory left in VCS after pair to pair communication for a certain time (e.g., 5 min) *fails*, the test case (T_2) verifying that the speed of fan in VCS locks near the speed set by the user always *fails* as well. This is in spite of the fact that all test cases are supposed to be executed independently of one another. Moreover, we noticed that when T_2 is

* Corresponding author at: Martin Linges vei 25, Fornebu, Norway.
E-mail address: dipesh@simula.no (D. Pradhan).

https://doi.org/10.1016/j.jss.2019.03.064
0164-1212/

executed as *pass*, another test case (T_3) that checks the CPU load measurement of VCS also always *passes*.

With this motivation, we recently proposed a black-box TP approach (named as REMAP) (Pradhan et al., 2018) to prioritize test cases dynamically based on the runtime execution results of the test cases using rule mining and search. The proposed approach (i.e., REMAP) consists of three key components: *Rule Miner* (*RM*), *Static Prioritizer* (*SP*), and *Dynamic Executor and Prioritizer* (*DEP*). First, *RM* defines *fail rules* and *pass rules* for representing the execution relations among test cases and mines these rules from the historical execution data using a rule mining algorithm (i.e., Repeated Incremental Pruning to Produce Error Reduction (RIPPER) (Cohen, 1995)). Second, *SP* defines two objectives: *Fault Detection Capability* (*FDC*) and *Test Case Reliance Score* (*TRS*), and applies a multi-objective search algorithm (i.e., Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al., 2002)) to statically prioritize test cases. Third, *DEP* executes the static prioritized test cases obtained from the *SP* and dynamically updates the test case order based on the runtime test case execution results together with the *fail rules* and *pass rules* from *RM*.

In this paper, we conduct an extensive empirical evaluation of REMAP using 1) three different rule mining algorithms: a) RIPPER, b) C4.5 (Quinlan, 1993), and c) Pruning Rule-Based Classification (PART) (Frank and Witten, 1998) together with 2) three different search algorithms: a) NSGA-II, b) Strength Pareto Evolutionary Algorithm (SPEA2) (Zitzler et al., 2001), and c) Indicator-based Evolutionary Algorithm (IBEA) (Zitzler and Künzli, 2004) for a total of nine different configurations of REMAP (i.e., three rule mining algorithms $\times$ three search algorithms). Additionally, we modify *SP* in REMAP by defining one additional objective (i.e., *Estimated Execution Time* (*EET*)) and statically prioritize with three objectives to check if the additional objective can improve the performance of REMAP for TP. Thus, we compare a total of 18 configurations of REMAP (i.e., nine configurations for REMAP with two objectives: *FDC* and *TRS* + nine configurations for REMAP with three objectives: *FDC, TRS*, and *EET*).

To empirically evaluate REMAP, we employed a total of five case studies (two industrial ones and three open source ones): 1) two data sets from Cisco related with VCS testing, 2) two open data sets from ABB Robotics for Paint Control (Spieker et al., 2017) and IOF/ROL (Spieker et al., 2017), and 3) one open Google Shared Dataset of Test Suite Results (GSDTSR) (Elbaum et al., 2014b). We compared the 18 configurations of REMAP with 1) two variants of random search (RS) (Elbaum et al., 2002; Kim and Porter, 2002; Elbaum et al., 2001): RS_{2obj} based on *FDC* and *TRS*, and RS_{3obj} based on *FDC, TRS*, and *EET*, 2) three prioritization approaches based on Greedy (Rothermel et al., 2001; Li et al., 2007; Hao et al., 2016): G_{1obj} based on *FDC*, G_{2obj} based on *FDC* and *TRS*, and G_{3obj} based on *FDC, TRS*, and *EET*, 3) 18 variants of search-based TP (SBTP) approaches (Li et al., 2013; Wang et al., 2016a; Parejo et al., 2016): nine each with two objectives and three objectives, and 3) six variants of rule-based TP (RBP) approaches (Anderson et al., 2014) (using the mined rules from *RM*): three each with two objectives and three objectives.

To assess the quality of the prioritized test cases, we use the Average Percentage of Faults Detected per Cost ($APFD_c$) metric. The results showed that the test cases prioritized by all the 18 variants of REMAP performed significantly better than RS for all the case studies, and the two best variants of REMAP with two objectives and three objectives performed significantly better than the best variants of the selected competing approaches by 84.4% and 88.9% of the case studies. Overall on average, the two best variants of REMAP with two objectives and three objectives managed to achieve on average 14.2% (i.e., 13.2%, 15.9%, 21.5%, 13.3%, and 6.9%) and 18.8% (i.e., 10.4%, 27.3%, 33.7%, 10.1%, and 12.2%) higher $APFD_c$ scores, respectively for the five case studies.

Note that this paper extends our previous work (Pradhan et al., 2018) with the following key contributions:

1) An extensive empirical evaluation of REMAP is performed by a) involving three rule mining algorithms together with three search algorithms for a total of nine configurations of REMAP and b) defining an additional objective for *SP* resulting in nine additional configurations of REMAP. The total evaluation included 18 configurations of REMAP, whereas our earlier work only included one configuration of REMAP.
2) Evaluation of REMAP is improved by involving 1) one additional variant of RS and Greedy, 2) 17 additional variants of SSBP approaches, and 3) five additional variants of RBP approaches.
3) The Average Percentage of Faults Detected per Cost ($APFD_c$) metric has been employed to evaluate the quality of the approaches to take into account the execution time of the test cases.

The remainder of the paper is organized as follows. Section 2 gives relevant background, and Section 3 motivates our work followed by a formalization of the TP problem in Section 4. Section 5 presents REMAP in detail, and Section 6 details the experiment design followed by presenting the results in Section 7. Overall discussion and threats to validity are presented in Section 8. Related work is presented in Section 9, and Section 10 concludes this paper.

2. Background

2.1. Data mining

Data mining is the process of extracting hidden correlations, patterns, and trends from large data sets that are both understandable and useful to the data owner (Hand, 2007). This process is achieved using pattern recognition technologies along with statistical and mathematical techniques (Larose, 2005). Data mining techniques have been widely applied to problems from different domains (e.g., science, engineering), and it is one of the fastest growing fields in the computer industry (Larose, 2005). Since all automated systems generate some form of data for diagnostic or analysis objectives, a large amount of data are produced (Aggarwal, 2015). However, the raw data might be collected from different formats, and the raw data might be unstructured and not immediately suitable for automated processing. Therefore, data mining consists of different phases such as data cleaning, feature extraction, and algorithmic design (Aggarwal, 2015). More specifically, the data cleaning and feature extraction phase converts the unstructured and complex data to a well-structured data set that can be effectively used by a computer program, which is then used by an algorithm to discover hidden patterns.

There are two different types of data mining methods: supervised learning (i.e., for labeled data) and unsupervised learning (i.e., for unlabeled data). Supervised learning aims to discover the relationship between input data (also called independent variables) and target attributes (also called dependent variable or outcome) (Maimon and Rokach, 2009). On the other hand, unsupervised learning aims to identify hidden patterns inside input data without labeled responses. More specifically, unsupervised learning groups instances without a pre-specified dependent attribute (Maimon and Rokach, 2009). In our context, we use labeled data to find potential rules regarding relations among the executions of test cases by using the past execution history as the training instances. Thus, we adopted supervised learning in our approach.

There are two main types of supervised models: classification models and regression models. Classifiers map the input space into predefined classes, while regressors map the input space into a real-valued domain (Maimon and Rokach, 2009). Since we

have predefined classes (i.e., test case execution results, e.g., *pass*), we use classification models. To construct classification rules, we selected three widely used rule mining algorithms, which employ different strategies to learn and mine rules from datasets (Frank and Witten, 1998; Witten and Frank, 2005). The used rule mining algorithms include C4.5 (Quinlan, 1993), Repeated Incremental Pruning to Produce Error Reduction (RIPPER) (Cohen, 1995), and Pruning Rule-Based Classification (PART) (Frank and Witten, 1998). Specifically, C4.5 takes a set of labeled data as input, creates a decision tree and tests it against unseen labeled test data for generalization. C4.5 employs a divide and conquer approach, and it is the most widely used rule mining algorithm in both the research community and industry (Witten and Frank, 2005; Wu et al., 2008). RIPPER employs the separate-and-conquer rule technique to generate rules directly from the training dataset, such that the rules are learned incrementally. RIPPER was designed to be fast and effective when dealing with large and noisy datasets as compared to decision trees (Cohen, 1995). While creating rules from decision tree is computationally expensive in the presence of noisy data, direct rule mining method has the over pruning (hasty generalization) problem (Holmes et al., 1999). PART employs separate-and-conquer to generate a set of rules and uses divide-and-conquer to build partial decision trees, such that a single rule is extracted for the branch that covers the maximum number of nodes (Holmes et al., 1999).

2.2. Multi-objective test prioritization

Multi-objective test prioritization aims to find tradeoff solutions for test case prioritization where various objectives (e.g., execution cost) conflict with one another, and no single optimal solution exist. Multi-objective test prioritization produces a set of solutions with equivalent quality (i.e., non-dominated solutions) based on *Pareto optimality*, which are called as Pareto fronts (Zitzler et al., 2000; Sayyad and Ammar, 2013; Srinvas and Deb, 1994). *Pareto optimality* defines the Pareto dominance to assess the quality of solutions (Wang et al., 2016b). Suppose a multi-objective TP problem consists of m objectives to be optimized, $O = \{o_1, o_2, \ldots, o_m\}$, and each objective can be measured using a fitness f_i from $F = \{f_1, f_2, \ldots, f_m\}$. If we aim to minimize the fitness function such that a lower value for an objective implies better performance, then solution s_a dominates solution s_b (i.e., $s_a \succ s_b$) *iff*:

$$\forall_{i=1,2,\ldots,m} f_i(s_a) \leq f_i(s_b) \wedge \exists_{i=1,2,\ldots,m} f_i(s_a) < f_i(s_b).$$

Multi-objective evolutionary algorithms have been widely applied for different multi-objective test prioritization problems. We employed three algorithms representative for multi-objective evolutionary algorithms for Pareto-dominance based approaches and indicator-based algorithms from the literature (Zhou et al., 2011; Jiang et al., 2015): Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al., 2002), Strength Pareto Evolutionary Algorithm (SPEA2) (Zitzler et al., 2001), and Indicator-based Evolutionary Algorithm (IBEA) (Zitzler and Künzli, 2004). These algorithms were selected since they are among the most known and recommended multi-objective evolutionary algorithms. Moreover, these three algorithms have been compared together in different studies (e.g., test data generation for feature testing of software product lines (Matnei Filho and Vergilio, 2015)) and shown promising results (Matnei Filho and Vergilio, 2015; Sayyad et al., 2013; Bringmann et al., 2011; Tran et al., 2013).

NSGA-II is based on *Pareto optimality*, and in NSGA-II, the solutions (i.e., chromosomes) in the population are sorted into several fronts based on the ordering of Pareto dominance (Deb et al., 2002). The individual solutions are selected from the non-dominated fronts, and if the number of solutions from the non-dominated front exceeds the specified population size, the solutions with a higher value of *crowding distance* are selected, where *crowding distance* is used to measure the distance between the individual solutions and the rest of the solutions in the population (Srinvas and Deb, 1994). SPEA2 is also based on *Pareto optimality*, and in SPEA2, the fitness value for each solution is calculated by summing up its raw fitness (based on the number of solutions it dominates) and density information (based on the distance between a solution and its nearest neighbors) (Zitzler et al., 2001). Initially, SPEA2 creates an empty archive and fills it with the non-dominated solutions from the population. In the subsequent generations, the solutions from the archive and the non-dominated solution are used to create a new population. If the number of non-dominated solutions is more than the maximum size of the specified population, the solution with the minimum distance to other solutions is selected by applying a *truncation operator*. IBEA uses performance indicators (e.g., *hypervolume* (*HV*)) instead of Pareto dominance to measure the quality of the solutions for multi-objective optimization (Zitzler and Künzli, 2004). Specifically, *HV* calculates the volume in the objective space covered by a non-dominated set of solutions (e.g., Pareto front) (Nebro et al., 2008). One potential downfall of using *HV* is the computational complexity of calculating the hypervolume measure as the number of objectives increase.

The greedy algorithm has also been widely employed to solve single and multi-objective test prioritization problem. Specifically, Greedy algorithms work on the "next best" search principle, such that the element (e.g., test case) with the highest weight (e.g., branch coverage) is selected first (Li et al., 2007). It is then followed by the element with the second highest weight and so on until all the elements have been selected or termination criteria of the algorithm are met (e.g., the total execution time of the selected test cases). The greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized (i.e., maximized or minimized). For multi-objective optimization, Greedy algorithm converts a multi-objective optimization problem into a single optimization problem using the weighted-sum method (Hajela and Lin, 1992) for assigning the fitness, such that each objective is given equal weight (i.e., if all the objective holds equal importance). After that, the weight of each element is obtained by adding the weighted objective values.

3. Motivating example

We have been collaborating with Cisco Systems, Norway for more than nine years to improve the quality of their video conferencing systems (VCSs) (Wang et al., 2013, 2014, 2016a). The VCSs are used to organize high-quality virtual meetings without gathering the participants to one location. These VCSs are used in many organizations such as IBM and Facebook. Each VCS has on average three million lines of embedded C code, and they are developed continuously. Testing needs to be performed each time the software is modified. However, it is not possible to execute all the test cases since there is only a limited time for execution and each test case requires much time for execution (e.g., 10 min). Thus, the test cases need to be prioritized to execute the important test cases (i.e., test cases most likely to fail) as soon as possible.

Through further investigation of VCS testing, we noticed that a certain number of test cases turned to *pass*/*fail* together, i.e., when a test case passes/fails some other test case(s) passed/failed almost all the time (with more than 90% probability). This is despite the fact that the test cases do not depend upon one another when implemented and are supposed to be executed independently. Moreover, the test engineers from our industrial partner are not aware of these execution relations (i.e., test case failing/passing together) when implementing and executing test cases. Note that we consider that each failed test case is caused by a separate fault since

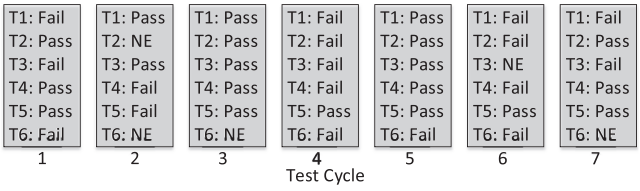


Fig. 1. Sample data showing the execution history for six test cases.*
*NE: Not Executed.

the link between actual faults and executed test cases are not explicit in our context, and this has been assumed in other literature as well (Elbaum et al., 2014a; Spieker et al., 2017; Anderson et al., 2014).

Fig. 1 depicts a running example to explain our approach with six real test cases from Cisco along with their execution results within seven test cycles. A test cycle is performed each time the software is modified, where a set of test cases from the original test suite are executed. When a test case is executed, there exist two types of execution results, i.e., *pass* or *fail. Pass* implies that the test case did not find any fault(s), while *Fail* denotes that the test case managed to detect a fault(s). From Fig. 1, we can observe that the execution of test cases T_1, T_2, T_3, T_4, T_5, and T_6 failed four, two, three, three, one, and four times, respectively within the seven test cycles. Moreover, we can observe that there exist certain relations between the execution results of some test cases. For example, when T_1 is executed as *fail*/*pass,* T_3 is always executed as *fail*/*pass* and vice versa, when both of them were executed.

Based on such observation, we can assume that internal relations for test case execution can be extracted from historical execution data, which can then be used to guide prioritizing test cases dynamically. More specifically, when a test case is executed as *fail* (e.g., T_1 in Fig. 1), the corresponding *fail* test cases (e.g., T_3 in Fig. 1) should be executed earlier as there is a high chance that the corresponding test case(s) is executed as *fail*. On the other hand, if a test case is executed as *pass* (e.g., T_4 in Fig. 1), the priority of the related *pass* test case(s) (T_2 in Fig. 1) need to be decreased (i.e., they need to be executed later). We argue that identifying such internal execution relations among test cases can help to facilitate prioritizing test cases dynamically, which is the key motivation of this work.

4. Basic notations and problem representation

This section presents the basic notations (Section 4.1) and problem representation (Section 4.2).

4.1. Basic notations

Notation 1. *TS* is an original test suite to be executed with n test cases, i.e., $TS = \{T_1, T_2, \ldots, T_n\}$. For instance, in the running example (Fig. 1), *TS* consists of six test cases.

Notation 2. *TC* is a set of p test cycles that have been executed, i.e., $TC = \{tc_1, tc_2, \ldots, tc_p\}$, such that one or more test cases are executed in each test cycle. For example, there are seven test cycles in the example (Fig. 1), i.e., p=7.

Notation 3. For a test case T_i executed in a test cycle tc_j, T_i has two possible verdicts, i.e., $v_{ij} \in \{pass, fail\}$. For example, in Fig. 1, the verdict for T_1 is *fail*, while the verdict for T_2 is *pass* in tc_1. If a test case has not been executed in a test cycle, it has no verdict. For instance, in Fig. 1, T_6 has no verdict (represented by *NE*) in tc_3. $V(T_i)$ is a function that returns the verdict of a test case T_i.

Notation 4. For a test case T_i executed in a test cycle tc_j, et_{ij} denotes the execution time of T_i in tc_j.

4.2. Problem representation

Let S represents all potential solutions for prioritizing *TS* to execute, $S = \{s_1, s_2, \ldots, s_q\}$, where $q = n!$. For instance in the running example with six test cases (Fig. 1), the total number of solutions $q = 720$. Each solution s_j is an order of test cases from *TS*, $s_j = \{T_{j1}, T_{j2}, \ldots, T_{jn}\}$ where T_{ji} refers to a test case that will be executed in the i^{th} position for s_j.

Test Case Prioritization (TP) Problem: TP problem aims to find a solution $s_f = \{T_{f1}, T_{f2}, \ldots, T_{fn}\}$ such that: $F(T_{fi}) \geq F(T_{fk})$, *where* $i < k \ \wedge \ 1 \leq i \leq n-1$ and $F(T_{fi})$ is an objective function that represents the fault detection capability of a test case in the i^{th} position for the solution s_f.

Dynamic TP Problem: The goal of dynamic TP is to dynamically prioritize test cases in the solution based on the runtime test case execution results to detect faults as soon as possible. For a solution $s_f = \{T_{f1}, T_{f2}, \ldots, T_{fn}\}$, the dynamic TP problem aims to update the execution order of the test cases in s_f to obtain a solution $s_f' = \{T_{f1}', \ldots, T_{fn}'\}$, such that $F(s_f') \geq F(s_f)$. $F(s_f')$ is a function that returns the value for the Average Percentage of Faults Detected per Cost for s_f'.

5. Approach: REMAP

This section first presents an overview of our approach for dynamic TP that combines rule mining and search in Section 5.1 followed by detailing its three key components (Sections 5.2–5.4).

5.1. Overview of REMAP

Fig. 2 presents an overview of REMAP consisting of three key components: *Rule Miner* (*RM*), *Static Prioritizer* (*SP*), and *Dynamic Executor and Prioritizer* (*DEP*). The core of *RM* is to mine the historical execution results of test cases and produce a set of execution relations (i.e., *Rules* depicted in Fig. 2) among test cases, e.g., if T_3 *fails* then T_1 *fails* in Section 3. Afterward, *SP* takes the mined rules and historical execution results of test cases as inputs to statically prioritize test cases for execution. Finally, *DEP* executes the statically prioritized test cases obtained from the *SP* one at a time and dynamically updates the order of the unexecuted test cases based on the runtime test case execution results (Fig. 2) to detect the fault(s) as soon as possible.

5.2. Rule miner (RM)

We first define two types of rules, i.e., *fail rule* and *pass rule* for representing the execution relations among test cases.

A fail rule specifies an execution relation between two or more test cases if a verdict of a test case(s) is linked to the *fail* verdict of another test case. The *fail rule* is formed as: $(V(T_i)\ AND, \ldots, \ AND\ V(T_k)) \xrightarrow{fail} (V(T_j) = fail) \wedge T_j \notin \{T_i, \ldots, T_k\}$, where $V(T_i)$ is a function that returns the verdict of a test case T_i (i.e., *pass* or *fail*). Note that for a *fail rule*, the execution relation must hold *true* for the specific test cases with more than 90% probability (i.e., confidence), a threshold that it is considered as being significant (Ordonez, 2006; Mosa and Yoo, 2013) and often used in the literature (Lin et al., 2002, 2000; Soto and Le Goues, 2018). For example, in Fig. 1, there exists a *fail rule* between T_1 and T_3: $(V(T_1) = fail) \xrightarrow{fail} (V(T_3) = fail)$, i.e., when T_1 has a verdict *fail* (executed as *fail*), T_3 always has the verdict *fail* and this rule holds *true* with 100% probability in the historical execution data (Fig. 1). With *fail rules*, we aim to prioritize and execute the test cases that are

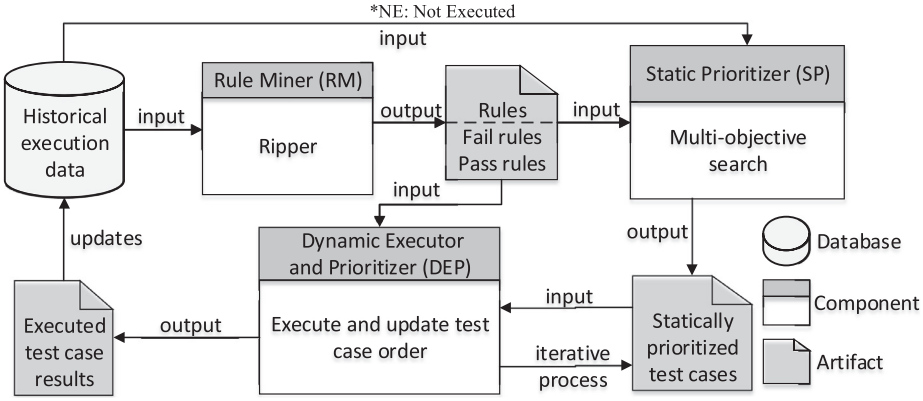


Fig. 2. Overview of REMAP.

Input: Set of test cycles $TC = \{tc_1, tc_2, \ldots, tc_p\}$, set of test cases $TS = \{T_1, T_2, \ldots, T_n\}$
Output: A set of *fail rules* and *pass rules*
Begin:

```
1  for (T ∈ TS) do                                  // for all the test cases
2     RuleSet ⟵ ∅
3     RuleSet ⟵ RMA(T, TC)                          // mine rules for T using TC
4     for (rule in RuleSet) do                      // for each mined rule
5        if (V(T) = fail) then                      // classify as fail rule
6           fail_rule ⟵ fail_rule ∪ rule
7        else if (V(T) = pass) then                 // classify as pass rule
8           pass_rule ⟵ pass_rule ∪ rule
9  return (fail_rule, pass_rule)
End
```

Fig. 3. Overview of rule miner.

more likely to *fail* as soon as possible. For instance, for the motivating example in Section 3, T_3 should be executed as early as possible when T_1 is executed as *fail*.

A pass rule specifies an execution relation between two or more test cases if a verdict of a test case(s) is linked to the *pass* verdict of another test case. The *pass rule* is in the form: $(V(T_i)\ AND, \ldots,\ AND\ V(T_k)) \overset{pass}{\rightarrow} (V(T_j) = pass) \wedge T_j \notin \{T_i, \ldots, T_k\}$ where $V(T_i)$ is a function that returns the verdict of test case T_i (i.e., *pass* or *fail*). Similarly, for a *pass rule*, the execution relation must hold *true* with more than 90% probability (i.e., confidence) as it is often used in the literature (Lin et al., 2002, 2000; Soto and Le Goues, 2018). For instance, in Fig. 1, there exists a *pass rule* between T_2 and T_4: $(V(T_4) = pass) \overset{pass}{\rightarrow} (V(T_2) = pass)$, i.e., when T_4 has a verdict *pass* (executed as *pass*), the verdict of T_2 is *pass* and this rule holds *true* with 100% probability based on the historical execution data (Fig. 1). With *pass rules*, we aim to de-prioritize the test cases that are likely to *pass* and execute them as late as possible. For instance, for the motivating example in Section 3, T_2 should be executed as late as possible when T_4 is executed as *pass*.

To mine *fail rules* and *pass rules*, our *RM* employs a rule mining algorithm (i.e., C4.5, RIPPER, or PART). More specifically, *RM* takes as input a test suite with n test cases, historical test case execution data (with a set of test cycles that lists the verdict of all the test cases). After that, *RM* produces a set of *fail rules* and *pass rules* as output. Fig. 3 demonstrates the overall process of *RM*. Recall that there are two possible verdicts for each executed test case (Section 4.1): *pass* or *fail*. For each test case in *TS*, *RM* uses the rule mining algorithm (e.g., RIPPER) to mine rules based on the historical execution data (i.e., *TC*) (lines 1–3 in Fig. 3). Afterward, the mined rules are classified into a *fail rule* or *pass rule* one at a time (lines 4–8 in Fig. 3) until all of them are classified. If the verdict of the particular test case is *fail*, the rule related to this test case is classified as a *fail rule* (lines 5–6 in Fig. 3); otherwise, the rule is classified as a *pass rule* (lines 7 and 8 in Fig. 3). In addition to the rules, rule mining algorithms output the support and violation for each rule. Based on this, we can calculate the probability that the test cases *passes*/*fails*. For instance, in Fig. 1, two rules are mined for T_1 using RIPPER: $(V(T_3) = fail) \overset{fail}{\rightarrow} (V(T_1) = fail)$ as a *fail rule* and $(V(T_3) = pass) \overset{pass}{\rightarrow} (V(T_1) = pass)$ as a *pass rule*. These rules held 100% true, and thus we can state that when T_3 *fails*/*passes* the probability that T_1 *fails*/*passes* is 100%. This process of rule mining and classification is repeated for all the test cases in the test suite (lines 1–8 in Fig. 3), and finally, the mined *fail rule* and *pass rule* are returned (line 9 in Fig. 3).

Table 1
Fail rule and *pass* rule from the motivating example using RIPPER.

#	*Fail rule*	#	*Pass rule*
1	$(V(T_1) = fail) \overset{fail}{\rightarrow} (V(T_3) = fail)$	5	$(V(T_1) = pass) \overset{pass}{\rightarrow} (V(T_3) = pass)$
2	$(V(T_2) = fail) \overset{fail}{\rightarrow} (V(T_4) = fail)$	6	$(V(T_3) = pass) \overset{pass}{\rightarrow} (V(T_1) = pass)$
3	$(V(T_3) = fail) \overset{fail}{\rightarrow} (V(T_1) = fail)$	7	$(V(T_4) = pass) \overset{pass}{\rightarrow} (V(T_2) = pass)$
4	$(V(T_4) = fail) \overset{fail}{\rightarrow} (V(T_2) = fail)$		

In this way, a set of *fail rules* and *pass rules* can be obtained from *RM*. For instance, in the motivating example (Section 3), we can obtain four *fail rules* and three *pass rules* for the six test cases from the seven test cycles (Fig. 1) using RIPPER as shown in Table 1.

5.3. Static Prioritizer (SP)

History-based TP techniques have been widely applied to prioritize the test cases based on their historical failure data, i.e., test cases that failed more often should be given higher priorities

(Kim and Porter, 2002; Khalilian et al., 2012; Spieker et al., 2017; Marijan et al., 2013; Pradhan et al., 2016). For example, T_1 and T_6 failed the highest number of times (i.e., four times) in Fig. 1, and therefore they should be given the highest priorities for execution out of the six test cases. Moreover, we argue that the execution relations among the test cases should also be taken into account for TP. The test cases whose results can be used to predict the results for more test cases (using *fail rules* and *pass rules*) need to be given higher priorities since their execution result can help predict the result of other test cases. For example in Fig. 1, the execution result of T_1 is related to one test case T_3, while the execution result of T_5 is not related to any test case as shown in Table 1. Thus, it is ideal to execute T_1 earlier than T_5 since based on the execution result of T_1, the related test case (i.e., T_3) can be prioritized (if T_1 *fails*) or deprioritized (if T_1 *passes*).

Thus, our *Static Prioritizer* (*SP*) defines two objectives: *Fault Detection Capability* (*FDC*) and *Test Case Reliance Score* (*TRS*), and uses multi-objective search to statically prioritize test cases before execution (as shown in Fig. 2). Note that in some context the execution time of the test cases is also available, which can be also be used for TP. Therefore, we have defined a third objective: *Estimated Execution Time* (*EET*) to use the execution time of the test cases, and we empirically evaluate if using three objectives can help obtain a better performance as compared to using two objectives. We formally define the objectives to guide the search toward finding optimal solutions in detail below.

Fault Detection Capability (*FDC*): The *FDC* for a test case is defined as the rate of failed execution of a test case in a given time period, and it has been frequently applied in the literature (Khalilian et al., 2012; Wang et al., 2016a). The *FDC* for a test case is calculated as: $FDC_{Ti} = \frac{Number\ of\ times\ that\ T_i\ found\ a\ fault}{Number\ of\ times\ that\ T_i\ was\ executed}$. *FDC* of T_i is calculated based on the historical execution information of T_i. For instance, in Fig. 1, the *FDC* for T_4 is 0.43 since it found fault three times out of seven executions. The *FDC* for a solution s_j can be calculated as: $FDC_{sj} = \frac{\sum_{i=1}^{n} FDC_{Ti} \times \frac{n-i+1}{n}}{mfdc}$, where, s_j represents any solution j (e.g., $\{T_1, T_4, T_3, T_6, T_2, T_5\}$ in Fig. 1), *mfdc* represents the sum of *FDC* of all the test cases in s_j, and n is the total number of test cases in s_j. Notice that a higher value of *FDC* implies a better solution. The goal is to maximize the *FDC* of a solution since we aim to execute the test cases that fail (i.e., detect faults) as early as possible.

Test Case Reliance Score (*TRS*): The *TRS* for a solution s_j is computed as: $TRS_{sj} = \frac{\sum_{i=1}^{n} TRS_{Ti} \times \frac{n-i+1}{n}}{otrps}$, where *otrps* represents the sum of *TRS* of all the test cases in s_j, n is the total number of test cases in s_j, and TRS_{Ti} is the test case reliance score (*TRS*) for the test case T_i. The *TRS* for a test case T_i is defined as the number of unique test cases whose results can be predicted by executing T_i using the defined *fail rules* and *pass rules* extracted from *RM* (Section 5.2). For instance, in Fig. 1, *TRS* for T_1 is 1 since the execution of T_1 can only be used to predict the execution result of T_3 based on the rule number 1 and 5 (Table 1) while *TRS* for T_5 is 0 as there is no test case that can be predicted based on the execution result of T_5. The goal is to maximize the *TRS* of a solution since we aim to execute the test cases that can predict the execution results of other test cases as early as possible.

Estimated Execution Time (*EET*): The *EET* of a test case is defined as the average execution time of the test case in a given time period. *EET* of a test case T_i executed m times in a given time period is computed as: $EET_{Ti} = \frac{\sum_{j=1}^{m} et_{ij}}{m}$, where et_{ij} denotes the execution time of T_i in a test cycle tc_j. For instance, in Fig. 1, if the execution time of T_1 for seven executions in the seven test cycles is 10.2 min (m), 10.8 m, 10.4 m, 9.6 m, 9.9 m, 10.3 m, 10.6 m; then the *EET* of T_1 is 10.3 m. The *EET* for a solution s_j can be calculated as: $EET_{sj} = \frac{\sum_{i=1}^{n} EET_{Ti} \times \frac{n-i+1}{n}}{\sum_{i=1}^{n} EET_{Ti}}$. Note that a lower value of *EET* implies a better solution. The goal is to minimize the *EET* of a solution since we aim to execute the test cases with lower execution time earlier than the test cases with higher execution time.

We further integrated these two objectives (i.e., *FDC* and *TRS*) and three objectives (i.e., *FDC, TRS*, and *EET*) with multi-objective search algorithms (i.e., IBEA, NSGA-II, or SPEA2). Note that *SP* produces a set of non-dominated solutions based on Pareto optimality (Zitzler et al., 2000; Sayyad and Ammar, 2013; Srinvas and Deb, 1994) with respect to the defined objectives. The Pareto optimality theory states that a solution s_a dominates another solution s_b if s_a is better than s_b in at least one objective and for all other objectives s_a is not worse than s_b (Sarro et al., 2016). Note that we randomly choose one solution from the generated non-dominated solutions as input for the component *Dynamic Executor and Prioritizer* (*DEP*) since all the solutions produced by *SP* have equivalent quality.

5.4. Dynamic Executor and Prioritizer (DEP)

The core of *DEP* is to execute the statically prioritized test cases obtained from *SP* (Section 5.3) and dynamically update the test case order using the *fail rules* and *pass rules* mined from *RM* (Section 5.2) as shown in Fig. 2. Thus, the input for *DEP* is a prioritization solution (i.e., static prioritized test cases) from *SP* and a set of mined rules (e.g., *fail rules*) from *RM*. The overall process of *DEP* is presented in Fig. 4, and it is explained using the example (Section 3) in Fig. 5.

Let us assume a prioritization solution is obtained as $\{T_1, T_4, T_3, T_6, T_2, T_5\}$ (Fig. 5) from *SP* for the test cases in the motivating example (Section 3). Moreover, seven *fail rules* and *pass rules* are extracted from *RM* as shown in Table 1. Initially, *DEP* checks if there are any test case(s) that always had the verdict *fail* or *pass* in the historical execution data. If there exists any such test case(s), *DEP* adds them to the set *AF* (if the verdict is always *fail*) or the set *AP* (if the verdict is always *pass*) (lines 2 and 3 in Fig. 4). For example in Fig. 1, T_6 always had the verdict *fail* in historical execution data, and thus T_6 is added to *AF*.

Afterward, *DEP* identifies the test cases to execute using the algorithm *Get-test-case-execution* (Algorithm 1). More specifically, *Get-test-case-execution* uses the statically prioritized solution, a set of always *fail* test cases *AF*, and a set of always *pass* test cases *AP*, to find the test case to execute. If there exists any test case in *AF* (line 1 in Algorithm 1), it is selected for execution (line 2 in Algorithm 1), removed from *AF* (line 3 in Algorithm 1), and returned (line 12 in Algorithm 1) to *DEP* (Fig. 4). For instance, initially, T_6 is selected for execution from *AF* in Fig. 5. Afterward, the selected test case (i.e., T_6) is executed (line 6 in Fig. 4), and based on the verdict of the executed test case, the *pass rule* or *fail rule* is employed if it exists (lines 7–12 in Fig. 4). Then, the executed test case is added to the final solution (line 13 in Fig. 4) and removed from the statically prioritized solution for execution (line 15 in Fig. 4). If there exists no related test case(s), the next test case is selected for execution (line 5 in Fig. 4).

To select the next test cases, Algorithm 1 is employed again. If there exists no more test case in *AF*, the first test case from the statically prioritized solution is selected for execution if it is not present in the set of *pass* test cases *AP* (lines 4–7 in). However, if a test case is present in *AP*, the next test case is selected from the static prioritized solution (lines 7–11 in Algorithm 1). For example, T_1 is executed after T_6 in Fig. 5 since it is the first test case from the statically prioritized solution obtained from *SP*, and it is not included in *AP*. Based on the verdict of the test case (line 6 in Fig. 4), *pass rule* or *fail rule* is employed once more (lines 7–12 in Fig. 4).

Input: Solution $s_f = \{T_{f1}, \ldots, T_{fn}\}$, *pass rules* PR, *fail rules* FR
Output: Dynamically prioritized test cases
Begin:
1 $k \longleftarrow 1$
2 $AF \longleftarrow AlwaysFailing(s_f)$ // set of test cases that always *fail*
3 $AP \longleftarrow AlwaysPassing(s_f)$ // set of test cases that always *pass*
4 **while** ($k \leq n$) and (termination_conditions_not_satisfied) **do**
5 $TE \longleftarrow Get-test-case-execution(s_f, AF, AP)$
6 $verdict \longleftarrow execute(TE)$ // execute the test case
7 **if** ($verdict$ = '*pass*') **then**
8 **if** ($TE \in PR$) **then** // if the test case has a *pass rule*
9 move the related test cases backward to the end of the solution
10 **else if** ($verdict$ = '*fail*') **then**
11 **if** ($TE \in FR$) **then** // if the test case has a *fail rule*
12 move the related test cases forward to execute next
13 $s_{final} \longleftarrow s_{final} \cup TE$ // dynamically prioritized solution
14 $k \longleftarrow k + 1$
15 remove TE from s_f // remove executed test case from s_f
16 return s_{final}
End

Fig. 4. Sample data showing the execution history for six test cases.

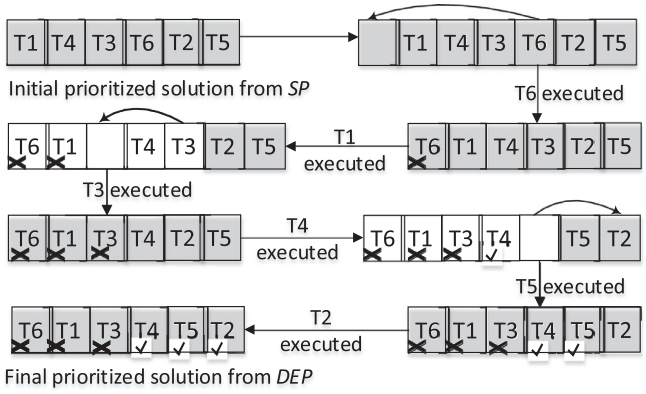


Fig. 5. Example of dynamic test case execution and prioritization.

If the test case has the verdict *fail* (line 10 in Fig. 4), it is checked if it is a part of any *fail rule* (line 11 in Fig. 4). If it is indeed a part of a *fail rule*, the related *fail* test case(s) is moved to the front of the solution s_f (line 12 in Fig. 4), e.g., T_3 is moved in front of T_4 in Fig. 5 since there exists a *fail rule* between T_1 and T_3.

Alternatively, if the test case has a verdict *pass*, it is checked if it is a part of any *pass rules* (line 7 in Fig. 4). If so, the related test case(s) is moved at the end of the solution. For example, T_2 is moved after T_5 in Fig. 5 since T_4 *passed* and there is a *pass rule* between T_4 and T_2 as shown in Table 1. This process of executing and moving the test case is repeated until all the test cases are executed (e.g., all six test cases are executed shown in Fig. 5) or the termination criteria (e.g., a predefined time budget) for the algorithm is met. The final solution lists the optimal order of the test cases that were executed. For example, the final execution order of these six test cases in Fig. 5 is: $\{T_6, T_1, T_3, T_4, T_5, T_2\}$ with three *fail* test cases: T_6, T_1, and T_3. Finally, at the end of the test cycle, the historical execution results of test cases are updated based on the verdicts of the executed test cases as shown in Fig. 2.

REMAP mines rules using historical execution results considering all the n test cycles. When a new test cycle $(n+1)$ is started, the mined rules (forming a rule set) together with the search algorithm are used to dynamically prioritize and execute the test cases. Then, after the $n+1$ test cycle is over, rule mining is performed by taking all the test case execution results of all the $n+1$ test cycles, results of which might lead to the updating of the already mined rule set. Additionally, the test cases can be run in parallel using REMAP. For instance, consider the initial prioritized solution in the paper is: $\{T_1, T_4, T_3, T_6, T_2, T_5\}$, and we have two rules if T_1 *fail* then T_5 *fail* and if T_4 *fail* then T_2 *fail*. Suppose, these two cases can be run in parallel, then T_1 and T_4 are executed first. If both the test cases are executed as *fail* then T_2 and T_5 are executed next, and finally, T_3 and T_6 are executed.

Note that REMAP includes *Static Prioritizer* (i.e., *SP* in Section 5.3) before *Dynamic Executor and Prioritizer* (i.e., *DEP* in Section 5.4) since we observed that not all the test cases can be associated with *fail rules* and *pass rules* (Section 5.2). For instance, in the motivating example (Section 3), the test case T_5 can be associated with no *fail rules* and *pass rules* (Section 5.2). Thus, as compared with randomly choosing a test case for execution when there is no mined rule to rely on, *SP* can help to improve the effectiveness of TP.

6. Empirical evaluation design

In this section, we present the experiment design (Table 2) with case studies (Section 6.1), research questions (Section 6.2), experiment tasks and evaluation metric (Section 6.3), statistical tests and experiment settings (Section 6.4).

6.1. Case studies

To evaluate the 18 variants of REMAP (i.e., 3 rule mining algorithms × 3 search algorithms × 2 set of objectives: two and three), we selected a total of five case studies: two case studies for two different VCS products from Cisco (i.e., CS_1 and CS_2), two open source case studies from ABB Robotics for Paint Control (CS_3) (Spieker et al., 2017) and IOF/ROL (CS_4) (Spieker et al., 2017), and Google Shared Dataset of Test Suite Results (GSDTSR) (CS_5) (Elbaum et al., 2014b). Note that the software in all the case studies is developed in a continuous integration environment. For instance, in the context of our industrial partner (CS_1 and CS_2), changes made by developers are merged in the VCS codebase daily. Testing is performed each time a new change is committed to the VCS codebase, i.e., a new test cycle starts. Thus, there is a one to one relationship between a test cycle and a software version. Specifically, each case study contains historical execution data of the test cases for more than 300 test cycles as shown in Table 3. The historical execution data of the test cases are used to mine execution relations using the *RM* (Section 5.2) and calculate the *FDC*

Table 2
Overview of the experiment design.

RQ	Task	Description	Algorithms	CS	Evaluation metric	Statistical tests	
1	$T_{1.1}$	Comparison of 18 variants of REMAP (with two objectives and three objectives) against RS_{2obj}	9 $REMAP_{2obj}$, 9 $REMAP_{3obj}$, RS_{2obj}	CS_1–CS_5	$APFD_c$	Mann–Whitney U Test	Vargha and Delaney $\hat{A}_{12}$
	$T_{1.2}$	Comparison of 18 variants of REMAP (with two objectives and three objectives) against RS_{3obj}	9 $REMAP_{2obj}$, 9 $REMAP_{3obj}$, RS_{3obj}				
2	$T_{2.1}$	Comparison of nine variants of REMAP with two objectives among each other	9 $REMAP_{2obj}$			Kruskal–Wallis Test, Dunn's test	
	$T_{2.2}$	Comparison of nine variants of REMAP with three objectives among each other	9 $REMAP_{3obj}$				
	$T_{2.3}$	Comparison of the two best variants of REMAP with two objectives and three objectives	$BestREMAP_{2obj}$, $BestREMAP_{3obj}$			Mann–Whitney U Test	
3	$T_{3.1}$	Comparison of the best variant of REMAP with two objectives against three variants of Greedy	$BestREMAP_{2obj}$, G_{1obj}, G_{2obj}, G_{3obj}				
	$T_{3.2}$	Comparison of the best variant of REMAP with three objectives against three variants of Greedy	$BestREMAP_{3obj}$, G_{1obj}, G_{2obj}, G_{3obj}				
4	$T_{4.1}$	Comparison of nine variants of SSBP with two objectives among each other	9 $SSBP_{2obj}$			Kruskal–Wallis Test, Dunn's test	
	$T_{4.2}$	Comparison of nine variants of SSBP with three objectives among each other	9 $SSBP_{3obj}$				
	$T_{4.3}$	Comparison of the best variant of REMAP with two objectives against the two best variants of SSBP with two objectives and three objectives	$BestREMAP_{2obj}$, $BestSSBP_{2obj}$, $BestSSBP_{3obj}$			Mann–Whitney U Test	
	$T_{4.4}$	Comparison of the best variant of REMAP with three objectives against the two best variants of SSBP with two objectives and three objectives	$BestREMAP_{3obj}$, $BestSSBP_{2obj}$, $BestSSBP_{3obj}$				
5	$T_{5.1}$	Comparison of three variants of RBP with two objectives	3 RBP_{2obj}			Kruskal–Wallis Test, Dunn's test	
	$T_{5.2}$	Comparison of three variants of RBP with three objectives	3 RBP_{3obj}				
	$T_{5.3}$	Comparison of the best variant of REMAP with two objectives against the best variants of RBP with two objectives and three objectives	$BestREMAP_{2obj}$, $BestRBP_{2obj}$, $BestRBP_{3obj}$			Mann–Whitney U Test	
	$T_{5.4}$	Comparison of the best variant of REMAP with three objectives against the best variants of RBP with two objectives and three objectives	$BestREMAP_{3obj}$, $BestRBP_{2obj}$, $BestRBP_{3obj}$				

Table 3
Overview of the case studies used for mining rules.

Case study	Data set	#Test cases	#Test cycles	#Verdicts	#Software versions	#Test cases fail
CS_1	Cisco Data Set1	60	8322	296,042	8322	14,890
CS_2	Cisco Data Set2	624	6302	149,039	6302	11,653
CS_3	ABB Paint Control	89	351	25,497	351	4952
CS_4	ABB IOF/ROL	1941	315	32,162	315	9283
CS_5	Google GSDTSR	5555	335	1253,464	335	3196

for each test case (Section 5.3). Afterwards, the experimental validation is conducted for the latest software version, whose results are not used for mining rules. For instance, CS_1 is evaluated based on test cycle number *8323*, CS_3 is evaluated based on test cycle number *352*. Note that the open source case studies for CS_3–CS_5 are publicly available at (Supplementary material, 2019).

6.2. Research questions

RQ1. Sanity Check: Is REMAP better than random search (RS) for all the five case studies? This research question is defined to check if our TP problem is non-trivial to solve. We assessed three rule mining algorithms in combination with three search algorithms for a total of nine different REMAP configurations while using 1) two objectives (*FDC* and *TRS*) and 2) three objectives for the search algorithms (*FDC, TRS*, and *EET*) defined in Section 5.3. The assessed REMAP configurations are: 1) $REMAP_{C4.5+IBEA(2obj)}$, 2) $REMAP_{C4.5+NSGA-II(2obj)}$, 3) $REMAP_{C4.5+SPEA2(2obj)}$, 4) $REMAP_{RIPPER+IBEA(2obj)}$, 5) $REMAP_{RIPPER+NSGA-II(2obj)}$, 6) $REMAP_{RIPPER+SPEA2(2obj)}$, 7) $REMAP_{PART+IBEA(2obj)}$, 8) $REMAP_{PART+NSGA-II(2obj)}$, 9) $REMAP_{PART+SPEA2(2obj)}$, 10) $REMAP_{C4.5+IBEA(3obj)}$, 11) $REMAP_{C4.5+NSGA-II(3obj)}$, 12) $REMAP_{C4.5+SPEA2(3obj)}$, 13) $REMAP_{RIPPER+IBEA(3obj)}$, 14) $REMAP_{RIPPER+NSGA-II(3obj)}$, 15) $REMAP_{RIPPER+SPEA2(3obj)}$, 16) $REMAP_{PART+IBEA(3obj)}$, 17) $REMAP_{PART+NSGA-II(3obj)}$, 18) $REMAP_{PART+SPEA2(3obj)}$. Moreover, we used RS with two objectives (i.e., RS_{2obj}) and three objectives (i.esfig0003., RS_{3obj}).

RQ2. Which configuration of REMAP performs the best while using two objectives and three objectives for the five case studies? Additionally, which is the best configuration of REMAP among the two objectives and three objectives? This research question aims to find the best configuration of REMAP among the nine configurations of REMAP while using 1) two objectives and 2) three objectives for the search algorithms. Additionally, we aim to find the best configuration of REMAP among the best configuration of REMAP with two objectives and three objectives.

RQ3. Is REMAP better than different variations of Greedy algorithm? This research question is defined to check if the best configuration of REMAP performs better than Greedy algorithm with one objective (G_{1obj}), two objectives (G_{2obj}), and three objectives (G_{3obj}). G_{1obj} prioritizes the test cases based on the objective *FDC*, G_{2obj} prioritizes the test cases based on the objectives *FDC* and *TRS* giving equal weight to the objectives, and G_{3obj} prioritizes the test cases based on *FDC, TRS*, and *EET* giving equal weight to each objective.

RQ4. Is REMAP better than the static search-based TP approach (SSBP), i.e., static prioritization solutions obtained from *SP* (Section 5.3)? This research question is defined to evaluate if dynamic prioritization (*DEP* in Section 5.4) can indeed help to improve the effectiveness of TP as compared with TP approaches without considering runtime test case execution results. More specifically, we compare the best configurations of REMAP with two objectives and three objectives against the best static search-based TP approaches with two objectives and three objectives. Note that there are also 18 different static search-based approaches while using two objectives and three objectives (similar as REMAP in RQ1) since the rules for the objective *TRS* are based on the specific rule mining algorithm (Section 5.3): 1) $SSBP_{C4.5+IBEA(2obj)}$, 2) $SSBP_{C4.5+NSGA-II(2obj)}$, 3) $SSBP_{C4.5+SPEA2(2obj)}$, 4) $SSBP_{RIPPER+IBEA(2obj)}$, 5) $SSBP_{RIPPER+NSGA-II(2obj)}$, 6) $SSBP_{RIPPER+SPEA2(2obj)}$, 7) $SSBP_{PART+IBEA(2obj)}$, 8) $SSBP_{PART+NSGA-II\ (2obj)}$, 9) $SSBP_{PART+SPEA2(2obj)}$, 10) $SSBP_{C4.5+IBEA(3obj)}$, 11) $SSBP_{C4.5+NSGA-II(3obj)}$, 12) $SSBP_{C4.5+SPEA2(3obj)}$, 13) $SSBP_{RIPPER+IBEA(3obj)}$, 14) $SSBP_{RIPPER+NSGA-II(3obj)}$, 15) $SSBP_{RIPPER+SPEA2(3obj)}$, 16) $SSBP_{PART+IBEA(3obj)}$, 17) $SSBP_{PART+NSGA-II(3obj)}$, 18) $SSBP_{PART+SPEA2(3obj)}$.

RQ5. Is REMAP better than the best rule-based TP approach (RBP), i.e., TP only based on the *fail* rules and *pass* rules from *RM* (Section 5.2)? Answering this research question can help us to know if *SP* together with *DEP* can help improve the effectiveness of TP as compared with the rule-based approach. Based on (Anderson et al., 2014), we designed rule-based TP approach (RBP) by applying the *RM* and *DEP* components. Note that the difference between RBP and our approach REMAP is that RBP uses the solutions produced by RS, i.e., RS_{2obj} and RS_{3obj} (from RQ1) as input rather than statically prioritized solutions from *SP*. Since there are three different rule-mining algorithms (e.g., C4.5) and two different versions of RS (e.g., RS_{2obj} and RS_{3obj}), there are a total of six configurations for RBP: 1) $RBP_{C4.5-2obj}$, 2) $RBP_{RIPPER-2obj}$, 3) $RBP_{PART-2obj}$, 4) $RBP_{C4.5-3obj}$, 5) $RBP_{RIPPER-3obj}$, and 6) $RBP_{PART-3obj}$. Therefore, we compare the best configurations of REMAP with two objectives and three objectives against the best configurations of RBP with two objectives and three objectives.

6.3. Experiment tasks and evaluation metric

6.3.1. Experiment tasks

To tackle RQ1, $T_{1.1}$ and $T_{1.2}$ are performed to compare the 18 variants of REMAP (i.e., using three rule mining algorithms and three search algorithms with two objectives and three objectives) against random search with two objectives (RS_{2obj}) and three objectives (RS_{3obj}) as shown in Table 2. For RQ2, $T_{2.1}$ and $T_{2.2}$ are performed to find the best variants of REMAP with two objectives and three objectives, respectively, and $T_{2.3}$ is done to compare the two best variants of REMAP with two objectives and three objectives. For RQ3, $T_{3.1}$ and $T_{3.2}$ are performed to compare the best variant of REMAP with two objectives and three objectives against the three variants of Greedy algorithms (i.e., G_{1obj}, G_{2obj}, and G_{3obj}), respectively as presented in Table 2. Furthermore, $T_{4.1}$ and $T_{4.2}$ are employed to find the best variants of SSBP with two objectives and three objectives, which are then compared against the best variant of REMAP with two objectives and three objectives using $T_{4.3}$ and $T_{4.4}$ to address RQ4 as shown in Table 2. Finally, to address RQ5, $T_{5.1}$ and $T_{5.2}$ are employed to find the best variant of RBP with two objectives and three objectives, which are then compared with the best variant of REMAP with two objectives and three objectives using $T_{5.3}$ and $T_{5.4}$.

For the industrial case studies (i.e., CS_1 and CS_2), we dynamically prioritize and execute the test cases in VCSs for the next test cycle (that has not been executed yet, e.g., test cycle 6303 for CS_2) and evaluate the different variants of the approaches: RS, Greedy, SSBP, RBP, and REMAP. However, for the open source case studies (i.e., CS_3–CS_5), we do not have access to the actual test cases to execute them as mentioned in Section 6.1. Thus, we use the historical test execution results without the latest test cycle for prioritizing the test cases and compare the performance of different approaches based on the actual test case execution results from the latest test cycle. For instance, for CS_5, which has execution results for 336 test cycles (Supplementary material, 2019), we used historical results from 335 test cycles to prioritize test cases for the next test cycle (i.e., 336). More specifically, we look at the execution result of each test case from the latest test cycle to know if the verdict of a test case is *fail* or *pass*, and accordingly, we apply the *fail rules* or *pass rules*. Note that such a way of comparison has been applied in the literature when it is difficult to execute the test cases at runtime in practice (Spieker et al., 2017; Anderson et al., 2014).

6.3.2. Evaluation metric

We used Average Percentage of Fault Detected per Cost metric ($APFD_c$) proposed by Elbaum et al. (2001) as the evaluation metric to compare the performance of different approaches. $APFD_c$ is an extended version of the Average Percentage of Faults Detected (*APFD*) metric (Rothermel et al., 1999) to consider the test case execution cost (e.g., execution time). Specifically, $APFD_c$ measures the effectiveness of a test case ordering by summing up the cost of the first test cases that can detect the faults, and it has been widely applied in the literature when test case cost is available (Di Nucci et al., 2015; Epitropakis et al., 2015; Luo et al., 2018). The $APFD_c$ for a solution can be calculated as: $APFD_c = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} et_j - \frac{1}{2}et_{TF_i})}{n \times m}$, where n denotes the number of test cases in the test suite *TS*, m denotes the number of faults detected by *TS*, et_j is the execution time of test case T_j in *TS*, and TF_i represents the first test case in the solution that reveals fault i. Note that the value of $APFD_c$ is between 0 and 1 (0 - 100%) and higher the $APFD_c$ score, lower the average cost is needed to detect the same number of faults.

6.4. Statistical tests and experiment settings

6.4.1. Statistical tests

Using the guidelines from (Arcuri and Briand, 2011), the Vargha and Delaney $\hat{A}_{12}$ statistics (Vargha and Delaney, 2000), Mann–Whitney *U* test (Mann and Whitney, 1947), Kruskal–Wallis test (Kruskal and Wallis, 1952), and Dunn's test (Dunn, 1964) with Bonferroni Correction (Bonferroni, 1936) are used to statistically evaluate the results for the five research questions as shown in Table 2. The Vargha and Delaney statistics is a non-parametric effect size measure and evaluates the probability of yielding higher values for the evaluation metric (i.e., $APFD_c$) for two approaches. Mann–Whitney *U* test is used to indicate whether the observations (i.e., $APFD_c$) in one data sample are likely to be larger than the observations in another sample. Kruskal–Wallis test indicates if there is a significant difference among the selected approaches. Dunn's test is based on rank sums and is used often (Dinno, 2015) as a post hoc procedure following rejection of a Kruskal–Wallis test to indicate which approach has a significant difference with which other approach. Specifically, for each pair of comparison, we used Vargha and Delaney $\hat{A}_{12}$ statistics as an effect size measure and Mann Whitney *U* Test to assess the statistical significance of the results. When comparing the different approaches with one another (e.g., ${}^9C_2$, i.e., 36 pairwise comparisons for REMAP with two objectives in $T_{2.1}$), we used Kruskal–Wallis test to evaluate if statistically significant difference exists among the approaches. After that, we used Dunn's test with Bonferroni Correction to evaluate in which pair statistical significant exists. Note that for Mann–Whitney, Kruskal–Wallis, and Dunn's test we chose the significance level of 0.05, i.e., a value less than 0.05 shows statistically significant differences.

Table 4
Parameter settings of the rule mining and search algorithms.

Algorithm	Parameter Settings
C4.5, PART	*Confidence threshold for pruning* = 0.25, *minimum number of instances per leaf* = 2, *number of folds for reduced error pruning* = 3, *seed used for randomizing the data* = 1
RIPPER	*Number of folds for pruning* = 3, *minimal total weight of the instances in a rule* = 2, *number of optimization runs* = 2, *seed used for randomizing the data* = 1, *pruning used* = True
NSGA-II, SPEA2	*Population size*: 100, *selection of parents*: binary tournament + binary tournament, *crossover type*: single-point, *crossover rate* = 0.9, *mutation type*: swap, *mutation rate* = 1.0/n
IBEA	*Population size*: 100, *selection of parents*: binary tournament + binary tournament, *crossover type*: single-point, *crossover rate* = 0.9, *mutation type*: swap, *mutation rate* = 1.0/n, *archive size*: 100

6.4.2. Experiment settings

We used the Weka data mining software (Hall et al., 2009) to implement the component *RM* (Section 5.2). The input data format to be used by our *RM* is a file composed of instances that contain test case verdicts such that each instance in the file represents a test cycle. Since the selection of the best parameter settings is application dependent, we used the default settings provided by Weka for all the three rule mining algorithms: C4.5, RIPPER, and PART. Table 4 presents the parameters used for configuring the three algorithms. In addition, we adopted the widely-used Java framework jMetal version 4.5.2 (Durillo and Nebro, 2011) to implement the component *SP* (Section 5.3) and the standard settings were applied (Table 4) to configure the search algorithms as such settings are usually recommended (Arcuri and Briand, 2011). For the search algorithms, the maximum number of fitness evaluations (i.e., the termination criteria of the algorithms) is set as 50,000. Additionally, each algorithm was run 30 times for each case study to account for random variations as suggested in (Arcuri and Briand, 2011). For the case studies, we encoded each test suite to be prioritized as an abstract format (i.e., JSON file), which contains the key information of the test cases for prioritization, e.g., test case *id, FDC*. Afterward, a search algorithm (e.g., NSGA-II) is employed to produce the prioritized test suite that is formed as the same abstract format (i.e., JSON file), which consists of a list of prioritized test cases. Finally, the prioritized test cases are selected from the original test suite using the test case *id* and put for execution. All the experiments were conducted on the Abel cluster at the University of Oslo (The Abel Computer Cluster, 2019).

7. Results and analysis

7.1. RQ1. Sanity check (18 variations of REMAP vs. RS_{2obj} and RS_{3obj})

Recall that RQ1 aims to assess the effectiveness of 18 variations of REMAP with two objectives (e.g., $REMAP_{C4.5+IBEA(2obj)}$) and three objectives (e.g., $REMAP_{PART+NSGA-II(3obj)}$) as compared to RS with two objectives (i.e., RS_{2obj}) and three objectives (i.e., RS_{3obj}) in terms of $APFD_c$. Using the Vargha and Delaney statistics and the Mann Whitney *U* test to analyze the results, we observed that all the 18 variations of REMAP performed significantly better than RS_{2obj} and RS_{3obj} for all the five case studies, i.e., $\hat{A}_{12}$ for the 18 variations of REMAP are greater than 0.7 and the *p*-values are less than 0.0001. The detailed results are presented in our technical report in (Technical Report, 2019).

Moreover, Table 5 presents the average $APFD_c$ scores produced by RS_{2obj} and RS_{3obj} for the five case studies. As compared to RS_{2obj} and RS_{3obj}, on average different variations of REMAP achieved a better $APFD_c$ score of 1) 25.8% by $REMAP_{C4.5+IBEA(2obj)}$, 2) 25.4% by $REMAP_{C4.5+NSGA-II(2obj)}$, 3) 25.3% by $REMAP_{C4.5+SPEA2(2obj)}$, 4) 26.8% by $REMAP_{RIPPER+IBEA(2obj)}$, 5) 26.9% by $REMAP_{RIPPER+NSGA-II(2obj)}$, 6) 26.8% by $REMAP_{RIPPER+SPEA2(2obj)}$, 7) 24.3% by $REMAP_{PART+IBEA(2obj)}$, 8) 23.7% by $REMAP_{PART+NSGA-II(2obj)}$, 9) 24.3% by $REMAP_{PART+SPEA2(2obj)}$, 10) 29.7% by $REMAP_{C4.5+IBEA(3obj)}$, 11) 26.3% by $REMAP_{C4.5+NSGA-II(3obj)}$, 12) 28.1% by $REMAP_{C4.5+SPEA2(3obj)}$, 13) 31.4% by $REMAP_{RIPPER+IBEA(3obj)}$, 14) 27.0% by $REMAP_{RIPPER+NSGA-II(3obj)}$, 15) 29.5% by $REMAP_{RIPPER+SPEA2(3obj)}$, 16) 28.9% by $REMAP_{PART+IBEA(3obj)}$, 17) 25.0% $REMAP_{PART+NSGA-II(3obj)}$, 18) 27.6% $REMAP_{PART+SPEA2(3obj)}$. The $APFD_c$ scores for the 18 variations of REMAP can be consulted from our technical report in (Technical Report, 2019). Fig. 6 presents the boxplot of $APFD_c$ produced by RS_{2obj} and RS_{3obj}.

Thus, we can answer RQ1 as REMAP can significantly outperform RS for all the five case studies defined in Section 6.1. Overall, on average, the different variants of REMAP with 18 different configurations achieved a better $APFD_c$ score of 26.8% as compared to RS_{2obj} and RS_{3obj}.

7.2. RQ2. Comparison of different variants of REMAP

This research question aims to find the best configuration of REMAP with two objectives and three objectives, and among one another using the $APFD_c$ score. The Kruskal–Wallis test was first performed for all the samples obtained by the 1) nine variants of REMAP with two objectives and 2) nine variants of REMAP with three objectives. We obtained *p*-values less than 0.0001 which shows that there exists at least one variant of REMAP with significant difference for each REMAP with two objectives and three objectives.

Therefore, for the post-hoc comparison we used the Vargha and Delaney statistics and Dunn's test with Bonferroni correction to rank the different variants of REMAP such that for two algorithms *A* and *B*, *A* is ranked higher than *B* if $\hat{A}_{12}$ is higher than 0.5 and the *p*-value is less than 0.05 or vice versa as mentioned in Section 6.4.1. If the *p*-value is higher than 0.05, the algorithms *A* and *B* are ranked in the same position. Specifically, our post-hoc analysis consists of ${}^9C_2 = 36$ combinations for each REMAP with two objectives and three objectives.

Table 6 shows the rank of different variants of REMAP with two objectives and three objectives such that a lower rank implies a better performance. Based on Table 6, we can observe that $REMAP_{RIPPER+SPEA2(2obj)}$ and $REMAP_{RIPPER+IBEA(3obj)}$ performed the best on average among the different variants of REMAP with two objectives and three objectives, respectively. The detailed results (i.e., $\hat{A}_{12}$ values, *p*-values, and $APFD_c$ scores for the 18 variations of REMAP) can be consulted from our technical report in (Technical Report, 2019).

Additionally, on comparing the best variant of REMAP with two objectives (i.e., $REMAP_{RIPPER+SPEA2(2obj)}$) against the best variant of REMAP with three objectives ($REMAP_{RIPPER+IBEA(3obj)}$), we noticed that $REMAP_{RIPPER+IBEA(3obj)}$ performed significantly better than $REMAP_{RIPPER+SPEA2(2obj)}$ for 60% (i.e., three case studies) as shown in Table 7. On average, $REMAP_{RIPPER+IBEA(3obj)}$ achieved an overall higher $APFD_C$ score of 4.6% as compared to $REMAP_{RIPPER+SPEA2(2obj)}$ for the five case studies as shown in Table 5. Moreover, Fig. 6 presents the boxplot of $APFD_C$ scores produced by $REMAP_{RIPPER+SPEA2(2obj)}$ and $REMAP_{RIPPER+IBEA(3obj)}$, which shows that overall $REMAP_{RIPPER+IBEA(3obj)}$ achieved a higher median $APFD_C$ score than $REMAP_{RIPPER+SPEA2(2obj)}$.

Table 5
APFD$_c$ scores in percentage for different approaches for the five case studies.

Approach	Case study					Approach	Case study				
	CS_1	CS_2	CS_3	CS_4	CS_5		CS_1	CS_2	CS_3	CS_4	CS_5
RS_{2obj}	53.37	51.86	48.91	51.99	54.59	$SSBP_{RIPPER+IBEA(3obj)}$	58.86	83.02	54.40	79.47	83.83
RS_{3obj}	52.87	51.79	54.49	51.49	52.42	$RBP_{RIPPER-2obj}$	69.09	66.91	64.48	83.94	78.47
G_{1obj}	61.61	35.50	38.81	94.22	89.36	$RBP_{RIPPER-3obj}$	69.25	66.80	68.67	83.86	78.37
G_{2obj}	68.04	62.54	21.45	88.11	90.13	$REMAP_{RIPPER+SPEA2(2obj)}$	76.21	77.03	67.63	90.91	84.27
G_{3obj}	68.04	64.70	21.45	88.11	90.10	$REMAP_{RIPPER+IBEA(3obj)}$	73.44	88.40	79.83	87.78	89.61
$SSBP_{RIPPER+SPEA2(2obj)}$	66.09	66.87	42.26	77.67	79.13						

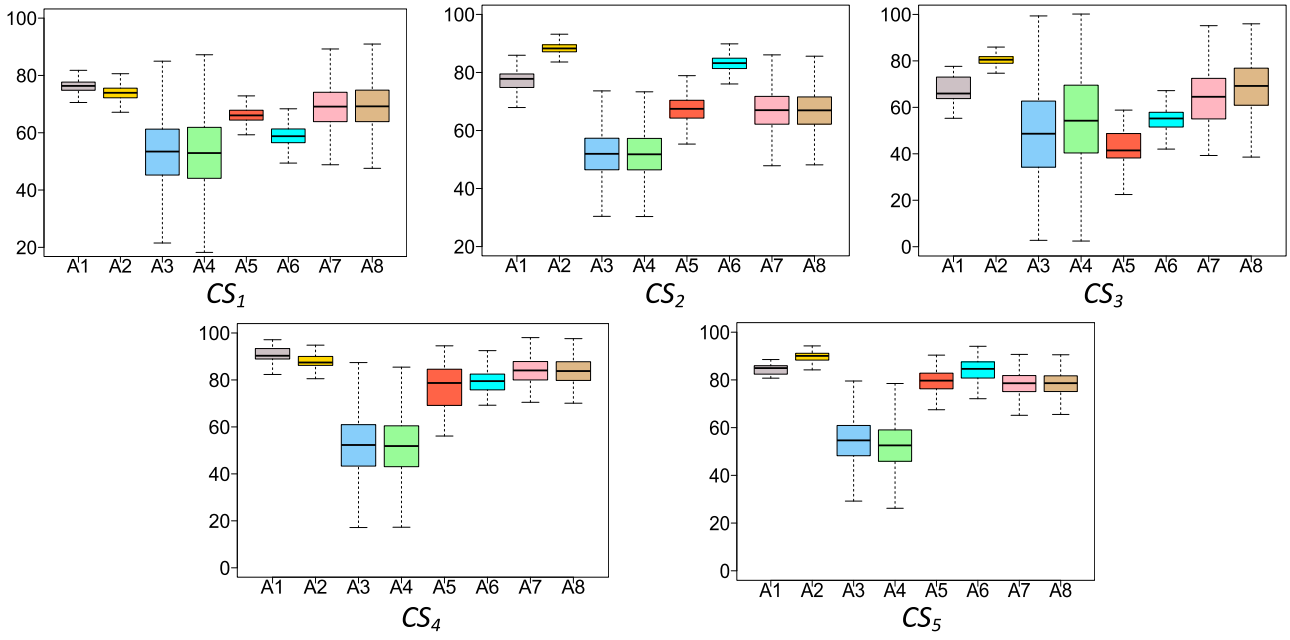


Fig. 6. Boxplot of *APFD$_c$* produced by different approaches for the five case studies*
*A1: $REMAP_{RIPPER+SPEA2(2obj)}$, A2: $REMAP_{RIPPER+IBEA(3obj)}$, A3: RS_{2obj}, A4: RS_{3obj}, A5: $SSBP_{RIPPER+SPEA2(2obj)}$, A6: $SSBP_{RIPPER+IBEA(3obj)}$, A7: $RBP_{RIPPER-2obj}$, A8: $RBP_{RIPPER-3obj}$.

Table 6
Ranking of 18 variants of REMAP for five case studies.[a]

CS	# of objectives	Rank									# of objectives	Rank								
		1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9
CS_1	2	R_6/R_4		R_5	R_1/R_2		R_3	R_8	R_7/R_9		3	R_4	R_5	R_6	R_1	R_2	R_3	R_8	R_9	R_7
CS_2		R_7	R_9	R_4	R_6/R_8		R_1/R_5		R_2/R_3			R_4	R_7	R_1	R_9	R_3	R_2	R_5	R_6	R_8
CS_3		R_6	R_5	R_4	R_3	R_9	R_1/R_7		R_2/R_8			R_4	R_7	R_1	R_6	R_3	R_9	R_5	R_2	R_8
CS_4		R_1	R_3	R_2	$R_5/R_6/R_7/R_8/R_9$					R_4		R_7	R_3	R_9	R_1	R_2	R_4	R_6	R_5	R_8
CS_5		R_5	R_3	R_2	R_1	$R_4/R_6/R_7/R_8/R_9$						R_9	R_4	R_6	R_1	R_3	R_7	R_8	R_2	R_5

[a] R_1: $REMAP_{C4.5+IBEA}$, R_2: $REMAP_{C4.5+NSGA-II}$, R_3: $REMAP_{C4.5+SPEA2}$, R_4: $REMAP_{RIPPER+IBEA}$, R_5: $REMAP_{RIPPER+NSGA-II}$, R_6: $REMAP_{RIPPER+SPEA2}$, R_7: $REMAP_{PART+IBEA}$, R_8: $REMAP_{PART+NSGA-II}$, R_9: $REMAP_{PART+SPEA2}$.

Table 7
Comparison of *APFD$_c$* for the two best variants of REMAP using the Vargha and Delaney Statistics and *U* test.[a]

Comparison	CS_1		CS_2		CS_3		CS_4		CS_5	
	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value
$REMAP_{4-3obj}$ vs. $REMAP_{6-2obj}$	0.23	Sig	1.00	Sig	0.97	Sig	0.24	Sig	0.92	Sig

[a] $REMAP_{4-3obj}$: $REMAP_{RIPPER+IBEA(3obj)}$, $REMAP_{6-2obj}$: $REMAP_{RIPPER+SPEA2(2obj)}$, Sig: <0.0001.

Thus, we can answer RQ2 as $REMAP_{RIPPER+SPEA2(2obj)}$, and $REMAP_{RIPPER+IBEA(3obj)}$ performed the best among the different variants of REMAP with two objectives and three objectives, respectively. Additionally, $REMAP_{RIPPER+IBEA(3obj)}$ performed the best on average among the 18 variants of REMAP with two objectives and three objectives.

7.3. RQ3. Comparison of the best variants of REMAP with Greedy

Recall that this research question aims to compare the performance of best variants of REMAP with two objectives (i.e., $REMAP_{RIPPER+SPEA2(2obj)}$) and three objectives (i.e., $REMAP_{RIPPER+IBEA(3obj)}$) against the three variants of Greedy: G_{1obj},

Table 8
Comparison of $APFD_c$ with respect to $REMAP_{RIPPER+SPEA2(2obj)}$ using the Vargha and Delaney Statistics and U test.[a]

RQ	Comparison	CS_1		CS_2		CS_3		CS_4		CS_5	
		$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
3	G_{1obj}	1.00	Sig	1.00	Sig	1.00	Sig	0.17	Sig	0.01	Sig
	G_{2obj}	1.00	Sig	1.00	Sig	1.00	Sig	0.85	Sig	0.01	Sig
	G_{3obj}	1.00	Sig	0.99	Sig	1.00	Sig	0.85	Sig	0.01	Sig
4	$SSBP_{RIPPER+SPEA2(2obj)}$	1.00	Sig	0.95	Sig	1.00	Sig	0.95	Sig	0.83	Sig
	$SSBP_{RIPPER+IBEA(3obj)}$	1.00	Sig	0.09	Sig	0.98	Sig	0.96	Sig	0.50	0.862
5	$RBP_{RIPPER-2obj}$	0.82	Sig	0.89	Sig	0.60	Sig	0.86	Sig	0.85	Sig
	$RBP_{RIPPER-3obj}$	0.80	Sig	0.90	Sig	0.46	Sig	0.86	Sig	0.86	Sig

[a] Sig: <0.0001.

Table 9
Comparison of $APFD_c$ with respect to $REMAP_{RIPPER+IBEA(3obj)}$ using the Vargha and Delaney Statistics and U test.[a]

RQ	Comparison	CS_1		CS_2		CS_3		CS_4		CS_5	
		$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
3	G_{1obj}	1.00	Sig	1.00	Sig	1.00	Sig	0.01	Sig	0.61	Sig
	G_{2obj}	0.92	Sig	1.00	Sig	1.00	Sig	0.43	Sig	0.48	**0.021**
	G_{3obj}	0.92	Sig	1.00	Sig	1.00	Sig	0.43	Sig	0.49	0.184
4	$SSBP_{RIPPER+SPEA2(2obj)}$	0.94	Sig	1.00	Sig	1.00	Sig	0.86	Sig	0.98	Sig
	$SSBP_{RIPPER+IBEA(3obj)}$	1.00	Sig	0.95	Sig	1.00	Sig	0.91	Sig	0.87	Sig
5	$RBP_{RIPPER-2obj}$	0.70	Sig	1.00	Sig	0.89	Sig	0.72	Sig	0.98	Sig
	$RBP_{RIPPER-3obj}$	0.69	Sig	1.00	Sig	0.82	Sig	0.72	Sig	0.98	Sig

[a] Sig: <0.0001.

G_{2obj}, and G_{3obj}. Table 8 presents the results of comparing $REMAP_{RIPPER+SPEA2(2obj)}$ against G_{1obj}, G_{2obj}, and G_{3obj}. Based on the results from Table 8, it can be observed that $REMAP_{RIPPER+SPEA2(2obj)}$ significantly outperformed the three variants of Greedy for 73.3% (11 out of 15) of the case studies. Moreover, we can observe from Table 5 that on average $REMAP_{RIPPER+SPEA2(2obj)}$ achieved a better $APFD_c$ score of 15.3%, 13.2%, and 12.7% as compared to G_{1obj}, G_{2obj}, and G_{3obj}.

Similarly, Table 9 presents the result of comparing $REMAP_{RIPPER+IBEA(3obj)}$ against G_{1obj}, G_{2obj}, and G_{3obj}, which shows that it significantly outperformed G_{1obj}, G_{2obj}, and G_{3obj} for 66.7% (10 out of 15) of the case studies and there was no significant difference in the performance for 6.7% (1 out of 15) of the case studies. Additionally, it can be observed from Table 5 that $REMAP_{RIPPER+IBEA(3obj)}$ attained a higher $APFD_c$ score of 19.9% (i.e., $\frac{(73.44-61.61)+(88.40-35.50)+(79.83-38.81)+(87.78-94.22)+(89.61-89.36)}{5}$), 17.8%, and 17.3% relative to G_{1obj}, G_{2obj}, and G_{3obj}.

Thus, we can answer RQ2 as the two best variants of REMAP with two objectives (i.e., $REMAP_{RIPPER+SPEA2(3obj)}$) and three objectives (i.e., $REMAP_{RIPPER+IBEA(3obj)}$) performed significantly better than G_{1obj}, G_{2obj}, and G_{3obj} for more than 65% of the case studies defined in Section 6.1. Overall, as compared to G_{1obj}, G_{2obj}, and G_{3obj}, $REMAP_{RIPPER+SPEA2(3obj)}$ and $REMAP_{RIPPER+IBEA(3obj)}$ achieved a better $APFD_c$ score of 13.7% and 18.3%, respectively.

7.4. RQ4. Comparison of the best variants of REMAP with the best variants of SSBP

This RQ aims to compare the best variant of REMAP with two objectives (i.e., $REMAP_{RIPPER+SPEA2(2obj)}$) and three objectives (i.e., $REMAP_{RIPPER+IBEA(3obj)}$) against the best variants of SSBP with two objectives and three objectives. First, Kruskal–Wallis test was first performed for all the samples obtained by the 1) nine variants of SSBP with two objectives and 2) nine variants of SSBP with three objectives. We obtained p-values less than 0.0001, which shows that there exists at least one variant of SSBP with a significant difference for each SSBP with two objectives and three objectives.

Table 10 shows the rank of nine variants of SSBP for each SSBP with two objectives and three objectives such that a lower rank implies a better performance using the Vargha and Delaney statistics and the Dunn's test with Bonferroni Correction as done in Section 7.2. Based on Table 10, we can observe that on average $SSBP_{RIPPER+SPEA2(2obj)}$ and $SSBP_{RIPPER+IBEA(3obj)}$ performed the best with two objectives and three objectives, respectively. The raw $\hat{A}_{12}$ values, p-values, and $APFD_c$ scores for the 18 variations of SSBP are provided in (Technical Report, 2019).

Based on Table 8, it can be observed that $REMAP_{RIPPER+SPEA2(2obj)}$ significantly outperformed $SSBP_{RIPPER+SPEA2(2obj)}$ and $SSBP_{RIPPER+IBEA(3obj)}$ for 80% (8 out of 10) of the case studies, while there was no significant difference for 10% (1 out of 10) of the case studies. Moreover, $REMAP_{RIPPER+SPEA2(2obj)}$ achieved a higher $APFD_c$ score of 12.8% and 7.3% on average as compared to $SSBP_{RIPPER+SPEA2(2obj)}$ and $SSBP_{RIPPER+IBEA(3obj)}$. Similarly, we can observe from Table 9 that $REMAP_{RIPPER+IBEA(3obj)}$ performed significantly better than $SSBP_{RIPPER+SPEA2(2obj)}$ and $SSBP_{RIPPER+IBEA(3obj)}$ for 100% (10 out of 10) of the case studies. Additionally, $REMAP_{RIPPER+IBEA(3obj)}$ obtained a better $APFD_c$ score of 17.4% and 11.9% as compared to $SSBP_{RIPPER+SPEA2(2obj)}$ and $SSBP_{RIPPER+IBEA(3obj)}$. Fig. 6 shows that the median $APFD_c$ scores produced by $REMAP_{RIPPER+SPEA2(2obj)}$, $REMAP_{RIPPER+IBEA(3obj)}$ are much higher than $SSBP_{RIPPER+SPEA2(2obj)}$ and $SSBP_{RIPPER+IBEA(3obj)}$.

Thus, we can answer RQ4 as the best variants of REMAP with two objectives, and three objectives managed to significantly outperform the best variants of SSBP for more than 80% of the case studies. Overall, as compared to the best variants of SSBP with two and three objectives, the best variants of REMAP with two objectives and three objectives achieved a better $APFD_c$ score of 10.1% and 14.7% on average.

7.5. RQ4. Comparison of the best variants of REMAP with the best variants of RBP

Recall that this RQ aims to compare the best variants of RBP using RS_{2obj} (i.e., $RBP_{RIPPER-2obj}$) and RS_{3obj} (i.e., $RBP_{RIPPER-3obj}$) against the best variants of REMAP with two objectives and three objectives obtained in Section 7.2. The Kruskal–Wallis test was first performed for all the samples obtained by the 1) three variants of

Table 10
Ranking of 18 variants of SSBP with two objectives and three objectives.[a]

CS	# of objectives	Rank									# of objectives	Rank								
		1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9
CS_1	2	S_6	S_4	S_5	S_2	S_1	S_3	S_7	S_8	S_9	3	S_4	S_5	S_6	S_2	S_7	S_9	S_8	S_1	S_3
CS_2		S_7	S_9	S_8	S_1	S_3	S_2	S_4	S_6	S_5		S_7	S_1	S_4	S_9	S_3	S_6	S_8	S_2	S_5
CS_3		S_6	S_9	S_3	S_5	S_2	S_8	S_1	S_4	S_7		S_3	S_9	S_4	S_6	S_7	S_2	S_1	S_8	S_5
CS_4		S_7	S_1	S_3	S_2	S_6	S_8	S_5	S_4	S_9		S_7	S_4	S_3	S_1	S_9	S_6	S_5	S_2	S_8
CS_5		S_5	S_2	S_6	S_8	S_3	S_4	$S_1/S_7/S_9$				S_1	S_3	S_6	S_9	S_7	S_4	S_2	S_5	S_8

[a] S_1: $SSBP_{C4.5+IBEA}$, S_2: $SSBP_{C4.5+NSGA-II}$, S_3: $SSBP_{C4.5+SPEA2}$, S_4: $SSBP_{RIPPER+IBEA}$, S_5: $SSBP_{RIPPER+NSGA-II}$, S_6: $SSBP_{RIPPER+SPEA2}$, S_7: $SSBP_{PART+IBEA}$, S_8: $SSBP_{PART+NSGA-II}$, S_9: $SSBP_{PART+SPEA2}$.

Table 11
Ranking of six variants of RBP with two objectives and three objectives.[a]

CS	Initial solution	Rank			Initial solution	Rank		
		1	2	3		1	2	3
CS_1	RS_{2obj}	A_2	A_1/A_3				A_5	A_4/A_6
CS_2		A_2	A_1	A_3	RS_{3obj}	A_5	A_4	A_6
CS_3		A_2	A_1/A_3				A_5	A_4/A_6
CS_4		A_2	A_1/A_3				A_5	A_4/A_6
CS_5		A_1/A_3		A_2		A_4/A_6		A_5

[a] A_1: $RBP_{C4.5-2obj}$, A_2: $RBP_{RIPPER-2obj}$, A_3: $RBP_{PART-2obj}$, A_4: $RBP_{C4.5-3obj}$, A_5: $RBP_{RIPPER-3obj}$, A_6: $RBP_{PART-3obj}$.

RBP with two objectives and 2) three variants of RBP with three objectives. We obtained *p*-values less than 0.0001, which shows that there exists at least one variant of RBP with a significant difference for each RBP with two objectives and three objectives.

We use the Vargha and Delaney statistics and the Dunn's test with Bonferroni Correction to rank the different variants of RBP (as done in Section 7.2) in Table 11 such that a lower rank implies better performance. We can observe from Table 11 that $RBP_{RIPPER-2obj}$ and $RBP_{RIPPER-3obj}$ performed the best with two objectives and three objectives. The detailed results (e.g., $APFD_c$ scores for the six variations of RBP) can be consulted from our technical report in (Technical Report, 2019).

Tables 8 and 9 presents the result of comparing the two best variants of REMAP with two objectives (i.e., $REMAP_{RIPPER+SPEA2(2obj)}$) and three objectives (i.e., $REMAP_{RIPPER+IBEA(3obj)}$) against the two best variants of RBP (i.e., $RBP_{RIPPER-2obj}$ and $RBP_{RIPPER-3obj}$). It can be observed from Tables 8 and 9 that as compared to $RBP_{RIPPER-2obj}$ and $RBP_{RIPPER-3obj}$, 1) $REMAP_{RIPPER+SPEA2(2obj)}$ performs significantly better for 90% (i.e., 9 out of 10) and 2) $REMAP_{RIPPER+IBEA(3obj)}$ performs significantly better for 100% (i.e., 10 out of 10) of the case studies. Furthermore, as observed from Table 5, $REMAP_{RIPPER+SPEA2(2obj)}$ and $REMAP_{RIPPER+IBEA(3obj)}$ achieved on average a better $APFD_c$ score of 6.2% and 10.8%, respectively. Moreover, the boxplot in Fig. 6 shows that the median $APFD_c$ score produced by $REMAP_{RIPPER+SPEA2(2obj)}$, $REMAP_{RIPPER+IBEA(3obj)}$ are higher than $RBP_{RIPPER-2obj}$, and $RBP_{RIPPER-3obj}$.

Therefore, we can answer RQ5 as the two best variants of REMAP with two objectives, and three objectives performed significantly better than the two best variants of RBP for more than 90% of the case studies. Overall on average, the best variants of REMAP with two objectives and three objectives achieved a better $APFD_c$ score of 6.2% and 10.8% as compared to the best variants of RBP with two objectives and three objectives.

7.6. Analysis of running time

Table 12 presents the average running time of the 18 variants of REMAP (i.e., 3 *rule mining algorithms* × 3 *search algorithms* × 2 *different set of objectives*) for the five case studies (i.e., CS_1 –CS_5). The running time is the sum of the time to mine rules (using the rule mining algorithm) and obtain solutions (using the search algorithms). Based on the results it can be observed that the running time of the two best variants of REMAP with two objectives (i.e., $REMAP_{RIPPER+SPEA2(2obj)}$) and three objectives (i.e., $REMAP_{RIPPER+IBEA(3obj)}$) is quite comparable with the other variants of REMAP with two objectives and three objectives, respectively. Overall, on average the two best variants of REMAP took 1.58, 2.79, 0.28, 11.60, and 103.89 min for CS_1, CS_2, CS_3, CS_4, and CS_5, respectively.

8. Overall discussion and threats to validity

8.1. Overall discussion

For RQ1, all the 18 variations of REMAP significantly outperformed the two variants of random search (i.e., RS_{2obj} and RS_{3obj}) for all the five case studies, which imply that the TP problems are not trivial to solve and require an efficient TP approach. Regarding RQ2, REMAP with the configuration of RIPPER and SPEA2 performed the best among the nine variants of RIPPER with two objectives, while REMAP with the configuration of RIPPER and IBEA performed the best among the nine variants of RIPPER with three objectives. This is in consistent with the performance of the individual algorithms in different approaches, for instance, 1) RIPPER performed the best among the three variants of RBP with both two objectives and three objectives (Section 7.5), and 2) RIPPER with SPEA2 and IBEA performed the best among the nine variants of SSBP with two objectives and three objectives, respectively (Section 7.4). Moreover, REMAP with the configuration of RIPPER and IBEA performed the best overall on average, which implies that using the execution time of the test cases as an objective can help improve TP. Therefore, if the execution time of the test cases is available, it is beneficial to use three objectives and REMAP with RIPPER and IBEA.

For RQ3, the best configuration of REMAP with two objectives and three objectives outperformed all the three variants of the Greedy algorithm (i.e., G_{1obj}, G_{2obj}, and G_{3obj}) for an average of 73.3% (11 out of 15) and 66.7% (10 out of 15) of case studies, respectively. The better performance of REMAP can be explained by the fact that Greedy algorithm greedily prioritizes the best test case one at a time in terms of the defined objectives until the

Table 12
Average running time of different variants of REMAP in minutes.[a]

Case Studies	# of objectives	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
CS_1	2	1.12	1.06	1.09	1.59	1.53	1.57	3.79	3.73	3.76
	3	1.12	1.08	1.11	1.60	1.52	1.59	3.80	3.72	3.79
CS_2	2	2.31	2.18	2.17	2.76	2.57	2.70	5.73	5.30	5.38
	3	2.01	1.86	1.96	2.87	2.34	2.49	5.45	5.02	5.19
CS_3	2	0.30	0.23	0.24	0.35	0.28	0.27	0.41	0.34	0.38
	3	0.31	0.22	0.34	0.28	0.26	0.29	0.40	0.34	0.39
CS_4	2	14.27	8.57	9.04	13.12	11.21	12.68	11.20	9.33	11.14
	3	12.31	5.29	6.56	10.53	8.50	10.18	13.06	6.28	8.42
CS_5	2	88.22	102.00	100.40	125.87	99.58	116.52	163.32	154.97	158.60
	3	57.67	75.83	65.07	91.27	70.00	96.52	102.62	114.52	103.33

[a] R_1: $REMAP_{C4.5+IBEA}$, R_2: $REMAP_{C4.5+NSGA-II}$, R_3: $REMAP_{C4.5+SPEA2}$, R_4: $REMAP_{RIPPER+IBEA}$, R_5: $REMAP_{RIPPER+NSGA-II}$, R_6: $REMAP_{RIPPER+SPEA2}$, R_7: $REMAP_{PART+IBEA}$, R_8: $REMAP_{PART+NSGA-II}$, R_9: $REMAP_{PART+SPEA2}$.

Algorithm 1 Get-test-case-execution.

Input: Solution $s_f = \{T_{f1}, \ldots, T_{fn}\}$, set of always *fail* test cases *AF*, set of always *pass* test cases *AP*
Output: A test case for execution
Begin:
1 **if** ($|AF| > 0$) **then**
2 $TE \leftarrow F(AF_0)$ // get the first *fail* test case
3 remove AF_0 from *AF* // remove the *fail* test case
4 **else** $TE \leftarrow F(s_{f0})$ // if there are not any test cases in *AF*
5 $move\_T \leftarrow True$ // initially test cases can be moved
6 $first\_T \leftarrow TE$
7 **while** (*TE* in *AP* and $move\_T$) **do**
8 move s_{f0} to end of s_f // move test case to the end
9 $TE \leftarrow F(s_{f0})$
10 **if** ($first\_T = TE$) **then**
11 $move\_T \leftarrow False$ // do not move test case anymore
12 return *TE*
End

termination conditions are met. However, the variants of Greedy may get stuck in local search space and result in sub-optimal solutions (Tallam and Gupta, 2006). Moreover, REMAP dynamically prioritizes the test cases based on the execution result of the test case using the mined rules defined in Section 5.2, which helps REMAP to prioritize and execute the faulty test cases as soon as possible while executing the test cases less likely to find fault later. On the contrary, for 26.7% of the case studies Greedy algorithm outperforms the best variants of REMAP. This can be explained by the fact that all test cases that failed in those case studies had a comparatively high value of *FDC* (e.g., >0.9 for CS_4), and the different variants of Greedy algorithm greedily prioritizes the test cases, e.g., G_{1obj} prioritizes the test cases based only on *FDC*. However, note that higher *FDC* does not always imply better $APFD_c$ as shown in Table 5 where the three variants of the Greedy algorithm have even worse performance than two variants of RS for CS_3.

Regarding RQ4, as compared to the best variants of SSBP with two objectives and three objectives, the best variant of REMAP with 1) two objectives performed significantly better for 80% and 2) three objectives performed significantly better for 100% of the case studies. This implies that it is essential to consider the runtime test case execution results in addition to the historical execution data when addressing TP problem. More specifically, the mined *fail rules* (Section 5.2) help to prioritize the related test cases likely to fail (to execute earlier) while the mined *pass rules* (Section 5.2) assist in deprioritizing the test cases likely to pass (to execute later). The best variant of REMAP with two objectives did not perform significantly better than the best variant of SSBP with three objectives for 20% of the case studies, which implies that the execution time of the test case needs to be considered for TP.

For RQ5, the best variant of SSBP with two objectives and three objectives significantly outperformed the best variants of RBP for 90% and 100% of the case studies. This can be explained by the fact that RBP has no heuristics to prioritize the initial set of test cases for execution and certain randomness is introduced when there are no mined rules that can be used to choose the next test cases for execution. As compared with RBP, REMAP uses *SP* (Section 5.3) to prioritize the test cases and take them as input for *Dynamic Executor and Prioritizer* (*DEP*). Therefore, the randomness of selecting the test cases for execution can be reduced when no mined rules can be applied.

Furthermore, Fig. 7 shows the percentage of faults detected when executing the test suite execution budget for the five case studies using the average of the best variants for the five approaches.: 1) RS (i.e., RS_{1obj} and RS_{2obj}), 2) Greedy (i.e., G_{1obj}, G_{2obj}, and G_{3obj}), 3) SSBP ($SSBP_{RIPPER+SPEA2(2obj)}$, $SSBP_{RIPPER+IBEA(3obj)}$), 4) RBP ($RBP_{RIPPER-2obj}$, $RBP_{RIPPER-3obj}$), and 5) REMAP ($REMAP_{RIPPER+SPEA2(2obj)}$, $REMAP_{RIPPER+IBEA(3obj)}$). As shown in Fig. 7, the two best variants of REMAP manage to detect the faults faster than the other approaches for almost all the case studies. On an average, the two best variants of REMAP only required 24% of the overall test suite execution budget to detect 82% of the faults for the five case studies while the best variants of RS, Greedy, SSBP, and RBP took 60%, 47%, 42%, and 36% of the overall test suite execution budget, respectively to detect the same amount of faults as detected by the two best variants of REMAP.

For CS_2, the mean $APFD_c$ produced by G_{1obj} is worse than RS_{2obj} and RS_{3obj} (Table 5) since the test cases that failed in CS_2 had a very low value for *FDC* (<0.2), i.e., the test cases did not fail many times in the past executions. Additionally, for CS_3, the mean $APFD_c$ obtained by G_{1obj}, G_{2obj}, G_{3obj} and the best variants of SSBP is worse than RS_{2obj} and RS_{3obj} (as shown in Table 5) due to the fact that the test cases that failed in CS_3 had also a low *FDC* (<0.5), and one test case that failed in CS_3 had the *FDC* value as 0. Thus, the best variants of SSBP, G_{1obj}, G_{2obj}, G_{3obj} are not able to prioritize that *fail* test case (with *FDC* value of 0) and it is executed very late (i.e., after executing more than 85% of the total test suite execution time) as shown in Fig. 7. However, the best variants of REMAP and

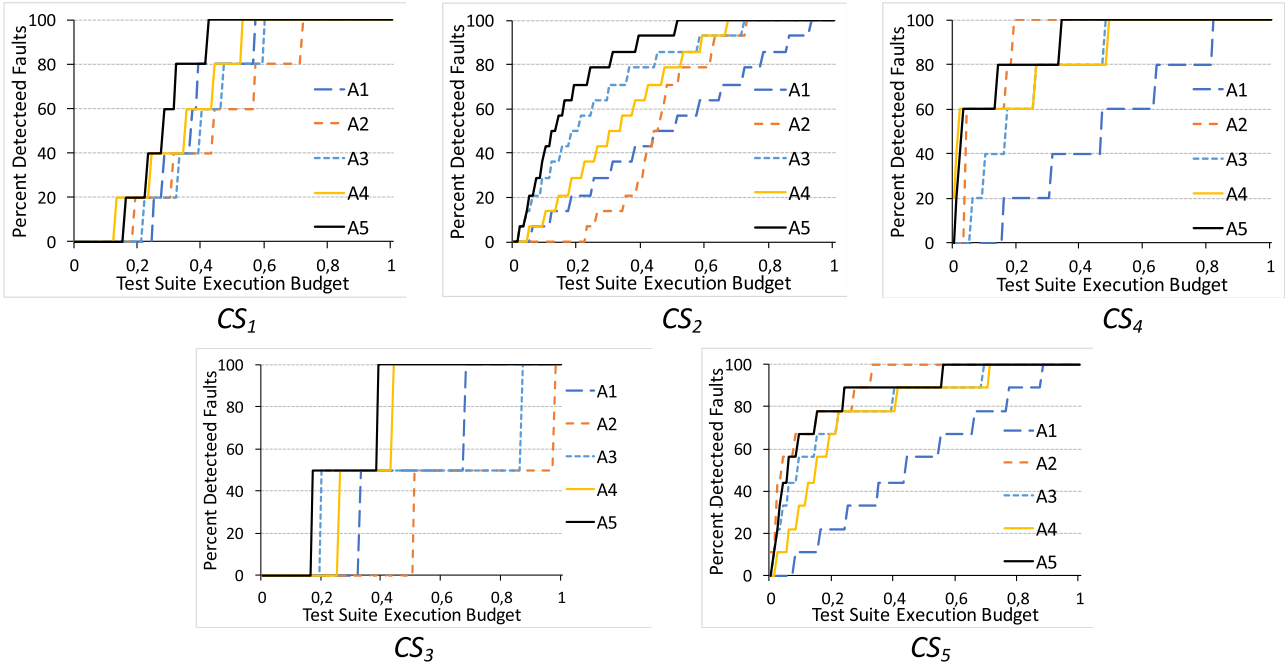


Fig. 7. Percentage of total faults detected on executing test suite total execution budget*.
A1: avg(RS_{2obj}, RS_{3obj}), A2: avg(G_{1obj}, G_{2obj}, G_{3obj}), A3: avg($SSBP_{RIPPER+SPEA2(2obj)}$, $SSBP_{RIPPER+IBEA(3obj)}$),
A4: avg($RBP_{RIPPER-2obj}$, $RBP_{RIPPER-3obj}$), A5: avg($REMAP_{RIPPER+SPEA2(2obj)}$, $REMAP_{RIPPER+IBEA(3obj)}$).

RBP still managed to find that *fail* test case by executing less than half the test suite execution budget. This is due to the fact that the mined rules help to deprioritize the test cases likely to *pass*, and thus, they are able to execute the test case for execution faster.

Regarding the limitations of REMAP, it is important to mention that if historical execution data is small, the application of REMAP is limited in practice. This is because REMAP requires historical execution data to mine the rules among test cases, and the accuracy of the mined rules increases with the size of the historical execution data. In addition, when test cases are run in parallel, it might decrease the efficiency of REMAP. However, to study the detailed effects on parallelization on REMAP, we need additional experiments, which we aim to perform in the near future. Additionally, the confidence value is very important to determine a rule to be a *true positive* or *true negative*. Specifically, the confidence value needs to be set based on the domain knowledge, case study, and thorough empirical evaluation. If we set the confidence value very low, more rules are generated and there is a high likelihood for the rule to be false positive, whereas if we set the confidence value very high, few rules are generated making it very difficult to use dynamic prioritization.

8.2. Threats to validity

The threats to *internal validity* consider the internal parameters (e.g., algorithm parameters) that might influence the obtained results (Runeson et al., 2012). In our experiments, *internal validity* threats might arise due to experiments with only one set of configuration settings for the algorithm parameters (de Oliveira Barros and Neto, 2011). However, note that these settings are from the literature (Arcuri and Fraser, 2011). Moreover, to mitigate the *internal validity* threat due to parameter settings of the rule mining algorithms (e.g., PART, RIPPER), we used the default parameter settings that have performed well in the state-of-the-art (Witten and Frank, 2005; Vijayarani and Divya, 2011; Ma et al., 2011; Song and Lee, 2017). We selected 90% or higher confidence value to mine rules since it is considered significant (Ordonez, 2006; Mosa and Yoo, 2013) and used in many works (Lin et al., 2002, 2000). Likewise, a recent paper (Soto and Le Goues, 2018) showed that rules with a confidence threshold of 90% managed to achieve a good tradeoff as compared to other confidence thresholds (50%, 60%, 70%, 80%, and 100%).

Threats to *construct validity* arise when the measurement metrics do not sufficiently cover the concepts they are supposed to measure (Li et al., 2007). To mitigate this threat, we compared the different approaches using the Average Percentage of Fault Detected per Cost metric ($APFD_c$) metric that has been widely employed in the literature when test case cost is available (Di Nucci et al., 2015; Epitropakis et al., 2015; Luo et al., 2018). Another threat to *construct validity* is the assumption that each failed test case indicates a different failure in the system under test. However, the mapping between fault and test case is not easily identifiable for both the two industrial case studies and three open source ones, and thus we assumed that each failed test case is assumed to cause a separate fault. Note that such assumption is held in many works in the literature (Elbaum et al., 2014a; Spieker et al., 2017; Anderson et al., 2014). Moreover, we did not execute the test cases from the three open source case studies since we do not have access to the actual test cases. Therefore, we looked at the latest test cycle to obtain the test case results, which have been done in the existing literature when it is challenging to execute the test cases (Spieker et al., 2017; Anderson et al., 2014).

The threats to *conclusion validity* relate to the factors that influence the conclusion drawn from the experiments (Wohlin et al., 2000). In our context, *conclusion validity* threat arises due to the use of randomized algorithms that is responsible for the random variation in the produced results. To mitigate this threat, we repeated each experiment 30 times for each case study to reduce the probability that the results were obtained by chance. Moreover, following the guidelines of reporting results for randomized algorithms (Arcuri and Briand, 2011), we employed the Vargha and Delaney statistics test statistics as the effect size measure to determine the probability of yielding higher performance by different

algorithms and Mann–Whitney *U* test for determining the statistical significance of results.

The threats to *external validity* are related to the external factors that affect the generalization of the results (Runeson et al., 2012). The first threat to *external validity* concerns the number of case studies used to verify the results. To mitigate this threat, we chose five case studies (two industrial ones and three open source ones) to evaluate REMAP empirically. It is also worth mentioning that such threats to *external validity* are common in empirical studies (Sarro et al., 2016). The second threat to *external validity* is the selection of rule mining and search algorithms. To mitigate this threat we selected three different rule mining algorithms from the three possible paradigms for supervised rule mining, and we also employed three representative multi-objective search algorithms from the literature (Zhou et al., 2011).

9. Related work

There exists a large body of research on TP (Rothermel et al., 1999; Li et al., 2007; Elbaum et al., 2001; Li et al., 2013; Walcott et al., 2006; Henard et al., 2016; Pradhan et al., 2017), and a broad view of the state-of-the-art on TP is presented in (Yoo and Harman, 2012; Catal and Mishra, 2013). Different kinds of literature have presented different prioritization techniques such as search-based (Li et al., 2007, 2013; Walcott et al., 2006) and linear programming based (Hao et al., 2016; Zhang et al., 2009; Mirarab et al., 2012). Since our approach REMAP is based on historical execution results and rule mining, we discuss the related work from these two angles.

9.1. History-Based TP (SP)

History-based prioritization techniques prioritize test cases based on their historical execution data with the aim to execute the test cases most likely to fail first (Kim and Porter, 2002). History-based prioritization techniques can be classified into two categories: static prioritization and dynamic prioritization. Static prioritization produces a static list of test cases that are not changed while executing the test cases while dynamic prioritization changes the order of test cases at runtime.

Static TP: Most of the history-based prioritization techniques produce a static order of test cases (Elbaum et al., 2014a; Kim and Porter, 2002; Khalilian et al., 2012; Noor and Hemmati, 2015; Marijan et al., 2013; Park et al., 2008; Fazlalizadeh et al., 2009). For instance, Kim and Porter (2002) prioritized test cases based on the historical execution data where they consider the number of previous faults exposed by the test cases as the key prioritizing factor. Elbaum et al. (2014a) used time windows from the test case execution history to identify how recently test cases were executed and detected failures for prioritizing test cases. Park et al. (2008) considered the execution cost of the test case and the fault severity of the detected faults from the test case execution history for TP. Wang et al. 2014,2016a) defined fault detection capability (*FDC*) as one of the objectives for TP while using multi-objective search. As compared to the above-mentioned works, REMAP poses at least three differences: 1) REMAP defines two types of rules: *fail rule* and *pass rule* (Section 5.2) and mines these rules from historical test case execution data with the aim to support TP; 2) REMAP defines a new objective (i.e., test case reliance score for the component *SP* in Section 5.3) to measure to what extent, executing a test case can be used to predict the execution results of other test cases, which is not the case in the existing literature; and 3) REMAP proposes a dynamic method to update the test case order based on the runtime test case execution results.

Dynamic TP: Qu et al. (Qu et al., 2007) used historical execution data to prioritize test cases statically and after that, the runtime execution result of the test case(s) is used for dynamic prioritization using the relation among test cases. To obtain relation among the test cases, they (Qu et al., 2007) group together the test cases that detected the same fault in the historical execution data such that all the test cases in the group are related. Our work is different from (Qu et al., 2007) in at least three aspects: 1) REMAP uses rule-mining to mine two types of execution rules among test cases (Section 5.2), which is not the case in (Qu et al., 2007); 2) sensitive constant needs to be set up manually to prioritize/deprioritize test cases in (Qu et al., 2007), however, REMAP does not require such setting; 3) REMAP uses either a) *FDC* and *TRS* or b) *FDC*, *TRS*, and *EET* (Section 5.3) for static prioritization unlike (Qu et al., 2007) whereas only *FDC* is considered for static prioritization in (Qu et al., 2007). We did not compare REMAP with this approach since we do not have detailed information about the faults (e.g., root cause of failure, type of failure) for both the open source and industrial case studies employed in this paper, which are required by the approach (Qu et al., 2007).

9.2. Rule mining for regression testing

There are only a few works that focus on applying rule mining techniques for TP (Mahali et al., 2016; Acharya et al., 2015). The authors in (Mahali et al., 2016; Acharya et al., 2015) modeled the system using Unified Modeling Language (UML) and maintained a historical data store for the system. Whenever the system is changed, association rule mining is used to obtain the frequent pattern of affected nodes that are then used for TP. As compared with these studies (Mahali et al., 2016; Acharya et al., 2015), REMAP is different in at least two ways: 1) REMAP uses the execution result of the test cases to obtain *fail rules* and *pass rules*; 2) REMAP dynamically updates the test order based on the test case execution results. Another work (Anderson et al., 2014) proposed a rule mining based technique for improving the effectiveness of the test suite for regression testing. More specifically, they use association rule mining to mine the execution relations between the smoke test failures (smoke tests refer to a small set of test cases that are executed before running the regression test cases) and test case failures using the historical execution data. When the smoke tests fail, the related test cases are executed. As compared to this approach, REMAP has at least three key differences: 1) we aim at addressing test case prioritization problem while the test case order is not considered in (Anderson et al., 2014); 2) REMAP uses a search-based test case prioritization component to obtain the static order of test case before execution, which is different than (Anderson et al., 2014) that executes a set of smoke test cases to select the test cases for execution; 3) REMAP defines two sets of rules (i.e., *fail rule* and *pass rule*) while only *fail rule* is considered in (Anderson et al., 2014).

In our conference paper (Pradhan et al., 2018), we proposed a TP approach, REMAP that uses rule mining (using RIPPER) and multi-objective search (using NSGA-II) for dynamic test case prioritization. The performance of REMAP is assessed using Average Percentage of Faults Detected (*APFD*) metric and compared with one variant of RS, two variants of Greedy, one variant of SSBP, and one variant of RBP. As compared to this work, our current paper has at least four key differences: 1) an extensive evaluation of REMAP is conducted using a combination of three rule mining algorithms (i.e., C4.5, RIPPER, and PART) and three search algorithms (i.e., NSGA-II, SPEA2, IBEA) for a total of nine different configurations of REMAP; 2) an additional objective (i.e., *EET*) for *SP* in REMAP is defined and, nine more configurations of REMAP are evaluated (i.e., in total we evaluate 18 configurations of REMAP); 3) the performance of the approaches are assessed using the Average Percentage of Faults Detected per Cost ($APFD_c$) metric that takes into account the execution time of the test cases; 4) different

variants of REMAP are compared with two variants of RS, three variants of Greedy, 18 variants of SSBP, and six variants of RBP.

10. Conclusion

This paper introduces and conducts an extensive empirical evaluation of a rule mining and search-based dynamic prioritization approach (named as REMAP) that has three key components (i.e., Rule Miner, Static Prioritizer, and Dynamic Executor and Prioritizer) with the aim to detect faults earlier. REMAP was extensively evaluated by employing five case studies: two industrial ones and three open source ones using 1) three rule mining algorithms, 2) three search algorithms, and 3) two different set of objectives (i.e., two and three). REMAP with the configuration of RIPPER and SPEA2 performed the best while using two objectives while REMAP with the configuration of RIPPER and IBEA performed the best while using three objectives among the 18 variations of REMAP. The results showed that the best variants of REMAP with two objectives and three objectives achieved a higher Average Percentage of Faults Detected per Cost ($APFD_c$) of 14.2% (i.e., 13.2%, 15.9%, 21.5%, 13.3%, and 6.9%) and 18.8% (i.e., 10.4%, 27.3%, 33.7%, 10.1%, and 12.2%) as compared to the best variants of random search, greedy, static search-based prioritization, and rule-based prioritization with two objectives and three objectives. In the future, we plan to involve test engineers from our industrial partner to deploy and assess the effectiveness of the best configuration of REMAP in real industrial settings. In addition, we want to experiment with different parameter settings of the algorithms to find the best parameter settings. Also, we plan to experiment with additional case studies and more number of test cycles to evaluate the robustness of the proposed approach in the near future.

Acknowledgments

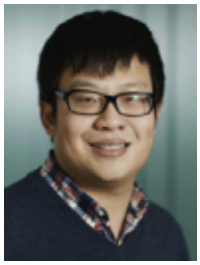
This research was supported by the Research Council of Norway (RCN) funded Certus SFI (grant no. 203461/O30). Tao Yue and Shaukat Ali are also supported by RCN funded Zen-Configurator project (grant no. 240024/F20) and RCN funded MBT4CPS project.

References

Acharya, A.A., Mahali, P., Mohapatra, D.P., 2015. Model based test case prioritization using association rule mining. In: Computational Intelligence in Data Mining, 3. Springer, pp. 429–440.
Aggarwal, C.C., 2015. Data Mining: The Textbook. Springer.
Anderson, J., Salem, S., Do, H., 2014. Improving the effectiveness of test suite through mining historical data. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 142–151.
Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE). IEEE, pp. 1–10.
Arcuri, A., Fraser, G., 2011. On parameter tuning in search based software engineering. In: Proceedings of the 3rd International Symposium on Search Based Software Engineering. Springer, pp. 33–47.
Bonferroni, C., 1936. Teoria statistica delle classi e calcolo delle probabilita. Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze 8, 3–62.
Borjesson, E., Feldt, R., 2012. Automated system testing using visual gui testing tools: a comparative study in industry. In: Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 350–359.
Bringmann, K., Friedrich, T., Neumann, F., Wagner, M., 2011. Approximation-guided evolutionary multi-objective optimization. In: IJCAI Proceedings-International Joint Conference on Artificial Intelligence, p. 1198.
Catal, C., Mishra, D., 2013. Test case prioritization: a systematic mapping study. Softw. Qual. J. 21 (3), 445–478.
Chittimalli, P.K., Harrold, M.J., 2009. Recomputing coverage information to assist regression testing. IEEE Trans. Softw. Eng. 35 (4), 452–469.
Cohen, W.W., 1995. Fast effective rule induction. In: Proceedings of the Twelfth International Conference on Machine Learning, pp. 115–123.
Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. 6 (2), 182–197 IEEE.
Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A., 2015. Hypervolume-based search for test case prioritization. In: International Symposium on Search Based Software Engineering. Springer, pp. 157–172.
Dinno, A., 2015. Nonparametric pairwise multiple comparisons in independent groups using Dunn's test. Stata J.
Dunn, O.J., 1964. Multiple comparisons using rank sums. Technometrics 6 (3), 241–252.
Durillo, J.J., Nebro, A.J., 2011. jMetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. 42 (10), 760–771 Elsevier.
Elbaum, S., Malishevsky, A., Rothermel, G., 2001. Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, pp. 329–338.
Elbaum, S., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. 28 (2), 159–182.
S. Elbaum, A. Mclaughlin, and J. Penix. (2014b). The Google Dataset of Testing Results. Available: https://code.google.com/p/google-shared-dataset-of-test-suite-results.
Elbaum, S., Rothermel, G., Penix, J., 2014a. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 235–245.
Epitropakis, M.G., Yoo, S., Harman, M., Burke, E.K., 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, pp. 234–245.
Fazlalizadeh, Y., Khalilian, A., Abdollahi Azgomi, M., Parsa, S., 2009. Incorporating historical test case performance data and resource constraints into test case prioritization. Tests Proofs 43–57.
Frank, E., Witten, I.H., 1998. Generating Accurate Rule Sets Without Global Optimization.
Hajela, P., Lin, C.-Y., 1992. Genetic search strategies in multicriterion optimal design. Struct. Optim. 4 (2), 99–107.
Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The WEKA data mining software: an update. ACM SIGKDD Explor. Newslett. 11 (1), 10–18.
Hand, D.J., 2007. Principles of data mining. Drug Saf. 30 (7), 621–622.
Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., Xie, T., 2016. To be optimal or not in test-case prioritization. IEEE Trans. Softw. Eng. 42 (5), 490–505.
Henard, C., Papadakis, M., Harman, M., Jia, Y., Le Traon, Y., 2016. Comparing white-box and black-box test prioritization. In: Proceedings of the 38th International Conference on Software Engineering (ICSE). IEEE, pp. 523–534.
Holmes, G., Hall, M., Prank, E., 1999. Generating rule sets from model trees. In: Australasian Joint Conference on Artificial Intelligence. Springer, pp. 1–12.
Jiang, S., Zhang, J., Ong, Y.-S., Zhang, A.N., Tan, P.S., 2015. A simple and fast hypervolume indicator-based multiobjective evolutionary algorithm. IEEE Trans. Cybern. 45 (10), 2202–2213.
Kaner, C., 1997. Improving the maintainability of automated test suites. Softw. QA 4 (4).
Khalilian, A., Azgomi, M.A., Fazlalizadeh, Y., 2012. An improved method for test case prioritization by incorporating historical test case data. Sci. Comput. Progr. 78 (1), 93–116.
Kim, J.-M., Porter, A., 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th International Conference on Software Engineering (ICSE). IEEE, pp. 119–129.
Kruskal, W.H., Wallis, W.A., 1952. Use of ranks in one-criterion variance analysis. J. Am. Statist. Assoc. 47 (260), 583–621.
Larose, D.T., 2005. Introduction to Data Mining. Wiley Online Library.
Li, Z., Bian, Y., Zhao, R., Cheng, J., 2013. A fine-grained parallel multi-objective test case prioritization on GPU. In: Proceedings of the International Symposium on Search Based Software Engineering. Springer, pp. 111–125.
Li, Z., Harman, M., Hierons, R.M., 2007. Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. 33 (4), 225–237 IEEE.
Lin, W., Alvarez, S.A., Ruiz, C., 2000. Collaborative recommendation via adaptive association rule mining. Data Mining and Knowledge Discovery 6, 83–105.
Lin, W., Alvarez, S.A., Ruiz, C., 2002. Efficient adaptive-support association rule mining for recommender systems. Data Min. Knowl. Discov. 6 (1), 83–105.
Luo, Q., Moran, K., Zhang, L., Poshyvanyk, D., 2018. How do static and dynamic test case prioritization techniques perform on modern software systems? An extensive study on GitHub projects. IEEE Trans. Softw. Eng..
Ma, B., Dejaeger, K., Vanthienen, J., Baesens, B., 2011. Software Defect Prediction Based on Association Rule Classification.
Mahali, P., Acharya, A.A., Mohapatra, D.P., 2016. Test case prioritization using association rule mining and business criticality test value. In: Computational Intelligence in Data Mining, 2. Springer, pp. 335–345.
Maimon, O., Rokach, L., 2009. Introduction to knowledge discovery and data mining. In: Data Mining and Knowledge Discovery Handbook. Springer, pp. 1–15.
Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. 50–60 JSTOR.
Marijan, D., Gotlieb, A., Sen, S., 2013. Test case prioritization for continuous regression testing: an industrial case study. In: IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 540–543.
Matnei Filho, R.A., Vergilio, S.R., 2015. A mutation and multi-objective test data generation approach for feature testing of software product lines. In: 2015 29th Brazilian Symposium on Software Engineering (SBES). IEEE, pp. 21–30.
Mirarab, S., Akhlaghi, S., Tahvildari, L., 2012. Size-constrained regression test case selection using multicriteria optimization. IEEE Trans. Softw. Eng. 38 (4), 936–956.

Mosa, A.S.M., Yoo, I., 2013. A study on PubMed search tag usage pattern: association rule mining of a full-day PubMed query log. BMC Med. Inf. Decis. Making 13 (1), 8.
Nebro, A.J., Luna, F., Alba, E., Dorronsoro, B., Durillo, J.J., Beham, A., 2008. AbYSS: adapting scatter search to multiobjective optimization. IEEE Trans. Evol. Comput. 12 (4), 439–457 IEEE.
Noor, T.B., Hemmati, H., 2015. A similarity-based approach for test case prioritization using historical failure data. In: Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 58–68.
de Oliveira Barros, M., Neto, A., 2011. Threats to validity in search-based software engineering empirical studies. UNIRIO-Universidade Federal do Estado do Rio de Janeiro, techreport 6.
Onoma, A.K., Tsai, W.-T., Poonawala, M., Suganuma, H., 1998. Regression testing in an industrial environment. Commun. ACM 41 (5), 81–86.
Ordonez, C., 2006. Comparing association rules and decision trees for disease prediction. In: Proceedings of the International Workshop on Healthcare Information and Knowledge Management. ACM, pp. 17–24.
Parejo, J.A., Sánchez, A.B., Segura, S., Ruiz-Cortés, A., Lopez-Herrejon, R.E., Egyed, A., 2016. Multi-objective test case prioritization in highly configurable systems: a case study. J. Syst. Softw. 122, 287–310.
Park, H., Ryu, H., Baik, J., 2008. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In: Proceedings of the Secure System Integration and Reliability Improvement. IEEE, pp. 39–46.
Pradhan, D., Wang, S., Ali, S., Yue, T., Liaaen, M., 2016. STIPI: using search to prioritize test cases based on multi-objectives derived from industrial practice. In: Proceedings of the International Conference on Testing Software and Systems. Springer, pp. 172–190.
Pradhan, D., Wang, S., Ali, S., Yue, T., Liaaen, M., 2017. CBGA-ES: a cluster-based genetic algorithm with elitist selection for supporting multi-objective test optimization. In: IEEE International Conference on Software Testing, Verification and Validation. IEEE, pp. 367–378.
Pradhan, D., Wang, S., Ali, S., Yue, T., Liaaen, M., 2018. REMAP: using rule mining and multi-objective search for dynamic test case prioritization. IEEE International Conference on Software Testing, Verification and Validation. IEEE.
Qu, B., Nie, C., Xu, B., Zhang, X., 2007. Test case prioritization for black box testing. In: Proceedings of the 31st Annual Computer Software and Applications Conference (COMPSAC). IEEE, pp. 465–474.
Quinlan, J.R., 1993. In: C4. 5: Programming for Machine Learning, 38. Morgan Kauffmann, p. 48.
Rothermel, G., Harrold, M.J., 1997. A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. (TOSEM) 6 (2), 173–210.
Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 1999. Test case prioritization: an empirical study. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, pp. 179–188.
Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 2001. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. 27 (10), 929–948.
Runeson, P., Host, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons.
Sarro, F., Petrozziello, A., Harman, M., 2016. Multi-objective software effort estimation. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp. 619–630.
Sayyad, A.S., Ammar, H., 2013. Pareto-optimal search-based software engineering (POSBSE): a literature survey. In: Proceedings of the 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. IEEE, pp. 21–27.
Sayyad, A.S., Menzies, T., Ammar, H., 2013. On the value of user preferences in search-based software engineering: a case study in software product lines. In: Proceedings of the 35th International Conference on Software Engineering. IEEE, pp. 492–501.
Song, K., Lee, K., 2017. Predictability-based collective class association rule mining. Expert Syst. Appl. 79, 1–7.
Soto, M., Le Goues, C., 2018. Using a probabilistic model to predict bug fixes. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 221–231.
Spieker, H., Gotlieb, A., Marijan, D., 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. ACM, pp. 12–22.
Srinvas, N., Deb, K., 1994. Multi-objective function optimization using non-dominated sorting genetic algorithms. Evol. Comput. 2 (3), 221–248 MIT Press.
Supplementary material. 2019 Available: https://remap-ICST.netlify.com.
Tallam, S., Gupta, N., 2006. A concept analysis inspired greedy algorithm for test suite minimization. ACM SIGSOFT Softw. Eng. Notes 31 (1), 35–42 ACM.
Technical Report 2019, (2018-02). Available: https://www.simula.no/publications/employing-rule-mining-and-multi-objective-search-dynamic-test-case-prioritization.
The Abel Computer Cluster. 2019. Available: http://www.uio.no/english/services/it/research/hpc/abel/.
Tran, R., Wu, J., Denison, C., Ackling, T., Wagner, M., Neumann, F., 2013. Fast and effective multi-objective optimisation of wind turbine placement. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1381–1388.
Vargha, A., Delaney, H.D., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. 25 (2), 101–132.
Vijayarani, S., Divya, M., 2011. An efficient algorithm for generating classification rules. Int. J. Comput. Sci. Technol. 2 (4).
Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S., 2006. Timeaware test suite prioritization. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, pp. 1–12.
Wang, S., Ali, S., Gotlieb, A., 2013. Minimizing test suites in software product lines using weight-based genetic algorithms. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1493–1500.
Wang, S., Ali, S., Yue, T., Bakkeli, Ø., Liaaen, M., 2016a. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In: Proceedings of the 38th International Conference on Software Engineering Companion. ACM, pp. 182–191.
Wang, S., Ali, S., Yue, T., Li, Y., Liaaen, M., 2016b. A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp. 631–642.
Wang, S., Buchmann, D., Ali, S., Gotlieb, A., Pradhan, D., Liaaen, M., 2014. Multi-objective test prioritization in software product line testing: an industrial case study. In: Proceedings of the 18th International Software Product Line Conference. ACM, pp. 32–41.
Witten, I.H., Frank, E., 2005. Data Mining: Practical machine Learning Tools and Techniques, Second Ed. Morgan Kaufmann, San Francisco, USA.
Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers 2000.
Wong, W.E., Horgan, J.R., London, S., Agrawal, H., 1997. A study of effective regression testing in practice. In: Proceedings of the Eighth International Symposium on Software Reliability Engineering. IEEE, pp. 264–274.
Wu, X., Kumar, V., Quinlan, J.R., Ghosh, J., Yang, Q., Motoda, H., et al., 2008. Top 10 algorithms in data mining. Knowl. Inf. Syst. 14 (1), 1–37.
Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22 (2), 67–120 Wiley Online Library.
Zhang, L., Hou, S.-S., Guo, C., Xie, T., Mei, H., 2009. Time-aware test-case prioritization using integer linear programming. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ACM, pp. 213–224.
Zhou, A., Qu, B.-Y., Li, H., Zhao, S.-Z., Suganthan, P.N., Zhang, Q., 2011. Multiobjective evolutionary algorithms: a survey of the state of the art. Swarm Evol. Comput. 1 (1), 32–49.
Zitzler, E., Deb, K., Thiele, L., 2000. Comparison of multiobjective evolutionary algorithms: empirical results. Evol. Comput. 8 (2), 173–195 MIT Press.
Zitzler, E., Künzli, S., 2004. Indicator-based selection in multiobjective search. In: International Conference on Parallel Problem Solving from Nature. Springer, pp. 832–842.
Zitzler, E., Laumanns, M., Thiele, L., 2001. SPEA2: improving the strength Pareto evolutionary algorithm. In: Proceedings of the Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems, pp. 95-100. Eidgenössische Technische Hochschule Zürich (ETH). Institut für Technische Informatik und Kommunikationsnetze (TIK).

Dipesh Pradhan is currently working as a research engineer at Simula Research Laboratory, Norway, and he is a Ph.D. student at the Department of Informatics, University of Oslo, Norway (2015 - present). He obtained his Master's degree in informatics: programming and networks from the University of Oslo in 2015. He has a couple of years of industrial experience as a software-testing engineer. His research interests include machine learning, search-based software engineering, test automation, cloud computing, and model-based engineering.



Shuai Wang is currently working as a senior test consultant at Testify AS (Norway) after successfully obtaining Ph.D. degree with an honor from the University of Oslo in 2015. He holds broad research interests such as searchbased software engineering, product line engineering, model-based testing, and empirical software engineering with more than 20 publications from wellrecognized international journals (such as JSS, EMSE and SOSYM) and highreputed international conferences (such as ICSE, MODELS, ISSRE, ICST, SPLC). He also a recipient of an ACM distinguished paper award of MODELS 2013 (also best application track paper) and an outstanding reviewer award during 2015–2016 of the Information and Software Technology journal.

Shaukat Ali is currently a chief research scientist in Simula Research Laboratory, Norway. His research focuses on devising novel methods for Verification and Validation (V&V) of large scale highly connected softwarebased systems that are commonly referred to as Cyber-Physical Systems (CPSs). He has been involved in several basic research, research-based innovation, and innovation projects in the capacity of PI/Co-PI related to Model-based Testing (MBT), Search-Based Software Engineering, and Model-Based System Engineering. He has rich experience of working in several countries including UK, Canada, Norway, and Pakistan. Shaukat has been on the program committees of several international conferences (e.g.,MODELS, ICST, GECCO, SSBSE) and also served as a reviewer for several software engineering journals (e.g., TSE, IST, SOSYM, JSS, TEVC). He is also actively participating in defining international standards on software modeling in Object Management Group (OMG), notably a new standard on Uncertainty Modeling.

Tao Yue is a chief research scientist at Simula Research Laboratory, Oslo, Norway and she is also affiliated with the University of Oslo. She has received the Ph.D. degree in the Department of Systems and Computer Engineering at Carleton University, Ottawa, Canada in 2010. Before that, she was an aviation engineer and system engineer for seven years. She has nearly 20 years of experience of conducting industry-oriented research with a focus on Model- Based Engineering (MBE) in various application domains such as Avionics, Maritime and Energy, Communications, Automated Industry, and Healthcare in several countries including Canada, Norway, and China. Her present research area is software engineering, with specific interests in Requirements Engineering, MBE, Model-based Testing, Uncertainty-wise Testing, Uncertainty Modeling, Search-based Software Engineering, Empirical Software Engineering, and Product Line Engineering, with a particular focus on large-scale software systems such as Cyber- Physical Systems. Dr. Yue has been on the program and organization committees of several international conferences (e.g., MODELS, RE, SPLC). She is also on the editorial board of Empirical Software Engineering. Dr. Yue is also actively participating in defining international standards in Object Management Group (OMG), including Precise Semantics for Uncertainty Modeling, SysML, and UTP.

Marius Liaaen started his career as HW designer with and designed ASIC's and PCB's used for computer clustering. Later in TANDBERG he worked on telecoms protocols implementation and testing followed by building up his testing team. He is very focused on test automation and currently his main interest is model based testing techniques.