UiO : University of Oslo
Faculty of mathematics and natural sciences

Evolutionary Computation Based Test Optimization of Large-Scale Systems

Dipesh Pradhan

Thesis submitted for the degree of Ph.D.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
December 2018

Abstract

Modern large-scale software systems are highly configurable. Thus, they require a large number of test cases to be implemented and revised for testing a variety of system configurations. This makes software testing of large-scale software systems very expensive and time-consuming. However, companies have limited time and resources to test software systems. They need to deliver high-quality software products while facing different constraints (e.g., time), which raises the need for cost-effective testing.

Driven by the needs of our industrial partner, Cisco Systems Norway for testing of video conferencing systems, this thesis applies a set of methods based on evolutionary computation for cost-effective testing. Specifically, cost-effective testing in Cisco can be formulated into four main problems: 1) test case prioritization (TCP) to cost-effectively prioritize the existing test cases, 2) test case selection (TCS) to cost-effectively select a subset of test cases from the test suite with maximum effectiveness and minimum cost, 3) test case implantation (TCI) to modify existing test cases to cost-effectively test the untested configurations, and 4) dynamic test case prioritization (DTP) to dynamically prioritize test cases based on runtime execution results of the test cases in order to further improve the results of test case prioritization.

To address the above-mentioned challenges, this thesis proposes a set of methods based on evolutionary computation. It includes a: 1) Search-based prioritization approach based on incremental unique coverage and positional impact, STIPI to address TCP (Paper A); 2) Search-based test case selection approach which can incorporate user preference for difference objectives to select test cases within a time budget to address TCS (Paper B); 3) Search-based test case implantation approach to automatically analyze and implant existing test cases, SBI to address TCI (Paper D); and 4) Test case prioritization approach, REMAP that uses rule mining and multi-objective search to dynamically prioritize test cases to address DTP (Paper E). In addition, this thesis proposes two new cluster-based genetic algorithms to address the shortcomings of the current state-of-the-art search algorithms in multi-objective test optimization (Paper C).

All the proposed methods have been extensively evaluated by comparing against the state-of-the-art approaches by employing different case studies (e.g., industrial, real-world, open source) and widely used evaluation metrics (e.g., average percentage of fault detected). The results showed that the proposed methods could significantly improve the performance for all the identified challenges, and thus, help in cost-effective testing.

Acknowledgements

Undertaking this PhD has been a truly life-changing experience for me. It would not have been possible to do without the guidance I received from many people.

Foremost, I would like to thank my supervisors: Shuai Wang, Shaukat Ali, and Tao Yue. Shuai, you have been a tremendous mentor for me. You have helped me develop both as a researcher and a person. Shaukat and Tao, both of you have been with me since my masters' thesis; your feedback has always been extremely helpful. I have learned from you how to do research, and I am very grateful for your guidance, continuous support, patience, and motivation.

Special acknowledgment to Certus for funding this PhD and providing me with an opportunity to conduct the research. I would also like to thank Simula Research Laboratory (SRL) and Simula School of Research and Innovation (SSRI) for providing an excellent workplace.

I owe my deepest gratitude to all the colleagues in the software engineering research field of Simula and several of my colleagues at Simula. In particular, to Safdar Aqeel Safdar and Carl Martin Rosenberg for all the good discussions at work and outside work. I will miss our whiteboard sessions.

I want to thank Marius Liaaen and his team members from Cisco Systems Norway. Through our several meetings, we identified interesting industrial problems for my thesis. In addition, they provided me with different industrial datasets for evaluating the proposed solutions, which significantly improved the quality of this thesis.

Last but not the least, I would like to thank my family and friends. To my parents, thanks for all their love and moral support throughout the years. And, to my beloved wife, Nabina, thank you for supporting me for everything and encouraging me throughout this experience. Your love is my inspiration and driving force.

List of Papers

My entire PhD work lead to in total of seven papers (four conferences and three journals). Since Paper C and Paper E are the journal extensions of Paper F and Paper G, these two papers (F and G) are not included in the thesis to avoid redundancy.

Paper A. STIPI: Using Search to Prioritize Test Cases Based on Multi-Objectives Derived from Industrial Practice

D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen.

In: *Proceedings of the 28th International Conference on Testing Software and Systems (ICTSS), pp. 172-190. Springer, 2016. DOI: 10.1007/978-3-319-47443-4_11.*

Paper B. Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study

D. Pradhan, S. Wang, S. Ali, and T. Yue.

In: *Proceedings of Genetic and Evolutionary Computation Conference (GECCO), pp. 1085-1092. ACM, 2016. DOI: 10.1145/2908812.2908850.*

Paper C. CBGA-ES+: A Cluster-Based Genetic Algorithm with Non-Dominated Elitist Selection for Supporting Multi-Objective Test Optimization

D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen.

In: *IEEE Transactions on Software Engineering (TSE), IEEE 2018. DOI: 10.1109/TSE.2018.2882176.*

Paper D. Search-Based Test Case Implantation for Testing Untested Configurations

D. Pradhan, S. Wang, T. Yue, S. Ali, and M. Liaaen.

Revision submitted to Journal of Information and Software Technology (IST), Elsevier.

Paper E. Employing Rule Mining and Multi-Objective Search for Dynamic Test Case Prioritization

D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen.

Submitted to Journal of Systems and Software (JSS), Elsevier.

Paper F. CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization

D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen.

In: *Proceedings of International Conference on Software Testing, Verification and Validation (ICST),* ***Nominated for the best paper award****, pp. 367-378. IEEE, 2017. DOI: 10.1109/ICST.2017.40.*

Paper G. REMAP: Using Rule Mining and Multi-Objective Search for Dynamic Test Case Prioritization

D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen.

In: *Proceedings of International Conference on Software Testing, Verification and Validation (ICST), pp. 46-57. IEEE, 2018. DOI: 10.1109/ICST.2018.00015.*

Contents

Part I

Summary

Summary

1 Introduction

Software is ubiquitous in our everyday life. It exists in our smartphones, computers, televisions, cars and so on. In order to deliver high-quality software, software testing is performed to reveal software faults and ensure that the execution of software matches its expected behavior [1]. Today software testing has become an integral part of software development accounting for up to 50% of development budgets [2-5]. The cost is even significantly higher for large-scale systems [6] since they are composed of diverse hardware and software, and a large number of configurations in these systems need to be tested. As a result, companies face different constraints (e.g., time, resources) while trying to deliver reliable and high-quality software to the market. This raises the need to test such systems cost-effectively by taking into consideration both the cost and effectiveness (e.g., fault detection capability) of testing.

In this thesis, we applied methods based on evolutionary computation to cost-effectively test large-scale systems based on our collaboration with Cisco Systems Norway, which develops video conferencing systems (VCSs) [7, 8]. These VCSs enable high-quality conference meetings. Based on our discussion with the test engineers from Cisco, we observed that cost-effectively testing VCSs is a challenging task since a large number of test cases have been designed and implemented. However, the test engineers have limited time and resources to execute the test cases. To address this issue, we formulated two types of multi-objective test optimization problems: 1) *test case prioritization*, which aims to prioritize the test cases in the test suite in an optimal order to maximize the effectiveness of the test cases (e.g., configuration coverage) and 2) *test case selection*, which aims to select a subset of test cases from the entire test suite that can be executed within the defined time budget with maximum effectiveness and minimum cost (e.g., execution time of each test case). As compared to *test case prioritization*, *test case selection* selects a subset of test cases, where the order of the test cases does not matter.

After addressing the multi-objective test optimization problems, we observed that the existing test cases in Cisco do not cover the entire configurations supported by the VCSs, and some test cases use the same configurations. Thus, end users can encounter unexpected errors when VCSs are used with untested configurations. Manually writing test cases to test the untested configurations requires a large amount of manual work, which is practically infeasible. Moreover, certain test cases verify similar configurations decreasing the efficiency of testing. To address this issue, we formulated a 3) *test case implantation* problem, which aims to modify existing test cases for testing the untested configurations cost effectively. With *test case implantation*, existing test cases are modified, and thus, runtime prioritization is required in order to refine the solutions for detecting the faults quicker. We termed this problem as 4) *dynamic test case prioritization*, which aims to dynamically prioritize the statically prioritized test cases based on the runtime execution result of the test cases. Fig. 1 presents the overall scope of the thesis.

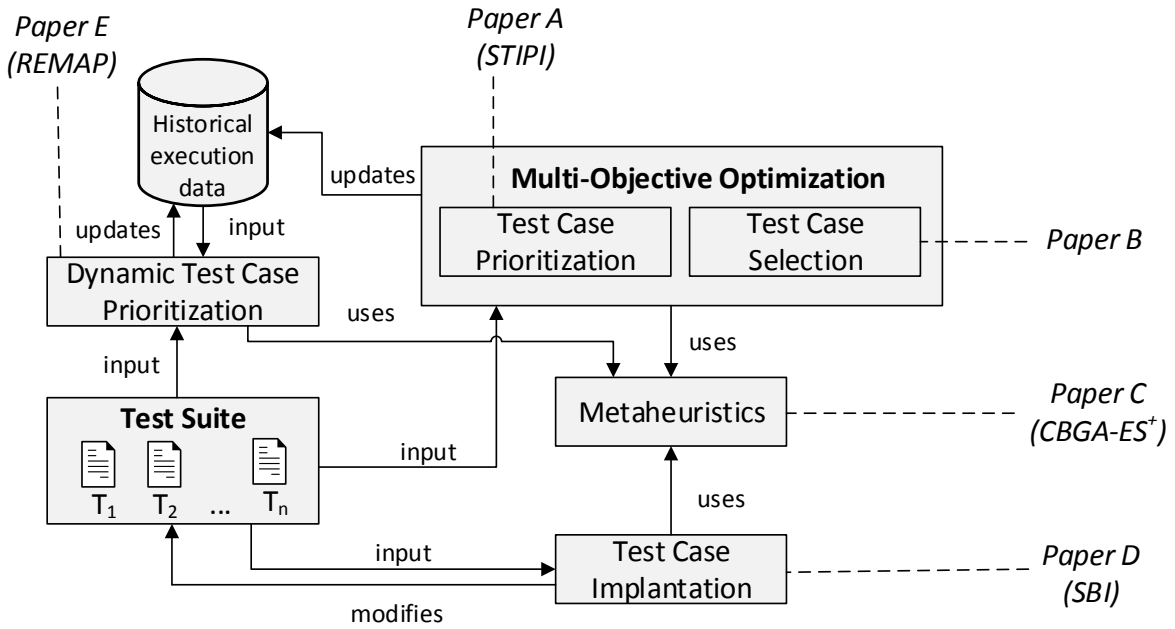

Fig. 1. The overall scope of the thesis

For the *test case prioritization* problem, the thesis proposes a search-based test case prioritization approach based on *incremental unique coverage* and *position impact* (STIPI). *Incremental unique coverage* was defined to consider only the incremental unique elements (e.g., test APIs) covered by a specific test case as compared with the elements covered by the already prioritized test cases. *Position impact* was defined to consider the impact of a specific test case on the quality of a prioritization solution, such that a test case with a higher execution position (i.e., scheduled to be executed earlier) has more impact than a test case with a lower execution position. Four effectiveness measures were defined to evaluate the quality of the solutions based on the configurations covered and fault detection capability of the test cases, and for each measure, a fitness function was defined. The results were empirically evaluated using three datasets from the industrial partner with four different time budgets: 25%, 50%, 75%, and 100% (i.e., no time budget). The results

showed that STIPI performed better than the five selected approaches from literature for on average 90% of the comparisons.

Concerning the *test case selection* problem, the thesis introduces a search-based multi-objective test case selection approach, which can incorporate user preference for different objectives. Four cost-effectiveness measures were defined to evaluate the quality of the solutions, and for each measure, a fitness function was defined. After that, a fifth fitness function was introduced to incorporate the user preference for different cost-effectiveness measures to guide Pareto-based search algorithms in a particular search space. The results were extensively evaluated using eight different search algorithms (consisting of three weight-based and five Pareto-based search algorithms) by employing a real-world case study and 10 artificial problems. The 10 artificial problems consisted of a different number of test cases and were created by simulating the real-world case study. Additionally, two different weight assignment strategies were used to assign the preference for different objectives for each search algorithm. The results showed that all the search algorithms with either of the weight assignment strategies significantly outperformed random search. In addition, Strength Pareto Evolutionary Algorithm (SPEA2) [9] with either of the weight assignment strategies performed the best among different search algorithms.

For the *test case implantation* problem, the thesis proposes a search-based test case implantation approach (named SBI) to automatically analyze and implant the existing test cases with the goal to test the untested configurations and test APIs. SBI consists of two key components: 1) *Test case analyzer* to obtain the program dependence graph for the statements in the test case by analyzing each test case; and 2) *Test case implanter* to select suitable test case for implantation using three operators: *selection*, *crossover*, and *mutation* and then implant (i.e., modify) the selected test cases using three operations: *addition*, *modification*, and *deletion*. Five cost-effectiveness measures were defined to assess the quality of the solutions (i.e., implanted test suites), and three variants of REMAP were empirically evaluated using an industrial case study and an open source case study. The results showed that all the three variants of SBI managed to cost-effectively improve the test suite. Additionally, SBI with Non-dominated Sorting Genetic Algorithm (NSGA-II) [10] performed the best. Specifically, SBI with NSGA-II achieved on average 19.3% higher coverage of configuration variable values for both the case studies.

Regarding the *dynamic test case prioritization* problem, the thesis proposes a test case prioritization approach named as REMAP that uses rule mining and multi-objective search to dynamically prioritize the test cases. REMAP consists of three key components: *Rule Miner (RM)*, *Static Prioritizer (SP)*, and *Dynamic Executor and Prioritizer (DEP)*. *RM* defines *fail rules* and *pass rules* for representing the execution relations among test cases and mines these rules from the historical execution data. *SP* defines objectives to guide the search and applies multi-objective search to prioritize the test cases statically. *DEP* executes the statically prioritized test cases obtained from the *SP* and dynamically updates the test cases order based on the runtime test case execution results, and *fail rules* and *pass rules* from *RM*. 18 variants of REMAP were empirically evaluated against 29 variants of 4

different approaches by employing two industrial and three open source case studies. The results showed that all the variants of REMAP performed significantly better than random search. In addition, the two best variants of REMAP (for two different sets of objectives) performed significantly better than the selected 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.

In addition to these works, the thesis proposes two *new cluster-based genetic algorithms with elitist selection* (CBGA-ES$^+$ and its predecessor, CBGA-ES in Paper F) to address the current shortcomings of the state-of-the-art search algorithms, which we observed while working on the above-mentioned multi-objective test optimization problems. Specifically, current multi-objective search algorithms make choices based on the random number generation when selecting parent solutions (i.e., stochastic parent solutions) to produce offspring solutions. However, if the selected parent solutions are suboptimal in the population, it might result in offspring solutions with bad quality, i.e., the produced offspring solutions have worse values for the different objectives as compared to the solutions in the population. Thus, stochastic parent selection may prevent algorithms in finding optimal solutions. To address this issue, the proposed *cluster-based genetic algorithms* employ an efficient elitist strategy to select the parent solutions for producing the offspring solutions. The results showed that CBGA-ES$^+$ performed significantly better than the state-of-the-art search algorithms and its predecessor, CBGA-ES for 66% of the experiments from five different multi-objective test optimization problems (e.g., *test case prioritization*). Moreover, for solutions in the same search space, CBGA-ES$^+$ managed to perform better than the selected search algorithms and its predecessor, CBGA-ES for 11.5% on average.

This thesis consists of two parts.

Summary: The first part of the thesis (Part I) consists of the following sections: Section 2 provides relevant background information required to understand the thesis. The research method is presented in Section 3. Section 4 briefly presents the contributions of the thesis followed by the summary of the key results in Section 5. Section 6 outlines future research direction, and Section 7 concludes the thesis.

Papers: The second part of the thesis (Part II) presents the published or submitted research papers, which are included in this thesis.

2 Background

This section provides the necessary technical background to understand the rest of this thesis. Section 2.1 provides a brief overview of software testing followed by an introduction to optimization problems in Section 2.2. Section 2.3 provides a short description of different search algorithms. Finally, Section 2.4 provides a brief introduction to data mining.

2.1 Software Testing

The goal of software testing is to verify that the software is reliable and works as expected. It is considered as one of the most important activities in the software development process. Studies show that software testing can consume up to 50% of the total software development costs [2-5]. Testing is used as a means of validation and verification, where the validation process describes whether the right software is built and the verification process describes if the software is built right [11]. Even though testing can be used to show the presence of errors, it cannot show that there are no errors [11]. Thus, the goal of testing is to find as many errors as possible, and this can help the stakeholders gain confidence in the correct functioning of the software in its intended environment.

Software testing can be classified into two main categories: black box testing and white box testing. Black box testing is also referred to as functional testing. In black box testing, a system under test (SUT) is considered as a black box, where the internals of the system are not known, i.e., only the input and the outputs of the SUT are known. Specifically, the test cases are generated from informal or formal specifications of the SUT, and then the outputs generated by the SUT are compared with the expected behavior as defined by the requirements/specifications. There exist different techniques for black box testing, such as boundary value analysis, equivalence partitioning, and pairwise testing. Black box testing is typically used when the source code is unavailable, or when testing the entire system.

White box testing is also referred to as structural testing. In white box testing, the software system is treated as a white box, and testing is based on the actual source code of the software. The test cases defined in white box testing have access to the algorithms and structures of the source code. White box testing aims to discover errors that have occurred during program implementation. Some examples of white box testing techniques include mutation testing, data-flow testing, path testing, and statement testing. White box testing is typically used during the early stages of the testing process where the programmer is in charge of executing the test suite.

2.2 Testing Optimization Problems

Several software testing problems (e.g., test case prioritization) can be reformulated as optimization problems, where the aim is to find the best solution(s) from the set of all feasible solutions corresponding to one or more given functions to be optimized. In order

to reformulate a software testing problem as an optimization problem, one needs to: 1) represent the problem so that it allows symbolic manipulation, 2) define a fitness function that captures the objective(s) to be optimized; and 3) employ a set of manipulation operators that allow changing the candidate solutions [12]. The representation of the candidate solutions depends on the nature of the problem. The fitness function is used to evaluate the quality of a candidate solution (i.e., an element in the search space), and thus guide the search in order to find the optimal solution. Some *operators* are used to mutate the candidate solutions while others are used to exchange part of their solutions to produce other candidate solutions. The optimization problem can be defined as either a maximization or minimization problem that provides the maximum or minimum value for the objective function within the search space.

If an optimization problem has only one objective function to be optimized, it is termed as a single-objective optimization problem. On the other hand, if an optimization problem involves multiple and often-conflicting objective functions to be optimized at the same time, it is termed as a multi-objective optimization problem. Single-objective optimization problems have only one solution while there might exist more than one best solution for multi-objective optimization problems since it is essential to analyze tradeoffs between the objectives. Therefore, for multi-objective optimization problems, a set of solutions with equivalent quality (i.e., non-dominated solutions) is usually produced based on *Pareto dominance* and *Pareto optimality* [13-15]. Specifically, *Pareto optimality* defines the *Pareto dominance* for assessing the quality of the solutions.

Suppose there are n objectives $O = \{o_1, o_2, \dots, o_n\}$ to be optimized for a multi-objective test optimization problem, and each objective can be measured using objective functions o_i from $F = \{f_1, f_2, \dots, f_n\}$. If we aim to minimize the objective function such that a lower value for an objective function implies better performance, then solution A *dominates* B (i.e., $A \succ B$) *iff*: $\forall_{i=1,2,\dots,n} f_i(A) \leq f_i(B) \wedge \exists_{i=1,2,\dots,n} f_i(A) < f_i(B)$. Additionally, solution A^* is *Pareto optimal* if it is not dominated by any other solution in the feasible region Ω, i.e., *iff* $A^* \succ C \; \forall \; C \neq A^* \in \Omega$. Specifically, Pareto optimal solutions are the solutions whose corresponding objective functions in F cannot be improved simultaneously, i.e., there does not exist other solution that can improve one of the objective functions without worsening other objective functions [16]. The non-dominated solutions are said to form a *Pareto-optimal set* and the corresponding *objective vectors* (with the values of the objective functions) form a *Pareto frontier*. In practice, a decision maker selects one of the solutions from the *Pareto frontier* based on his/her preference of the objective functions.

Fig. 2 presents a graphical representation for a two objective minimization problem (i.e., a lower value for the objective is better). In Fig. 2, A dominates B, C, and D since the values of both the objectives (i.e., *min* F_1 and *min* F_2) for A is lower than B, C, and D. Additionally, E and A are non-dominated solutions since E is better than A for the objective function *min* F_1 while A is better than E for the objective function *min* F_2, and there does not exist any other solution in Fig. 2 that has better values for both the objectives simultaneously. Thus, E and A can be used to form a *Pareto front*.

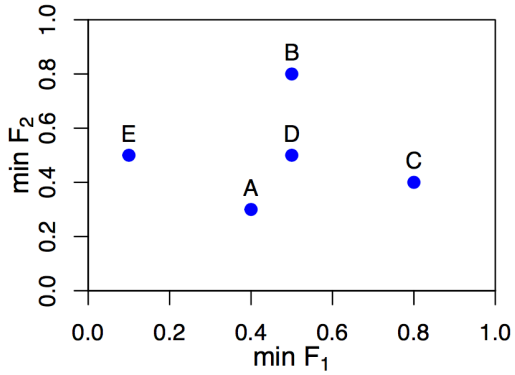

Fig. 2. Pareto dominance for a two objective minimization problem

When the search space becomes too large, specific techniques are required to solve both the single-objective and multi-objective test optimization problems in a reasonable time. To address this issue, Search Based Software Testing (SBST) employs metaheuristics to tackle them efficiently after reformulating the testing problems as optimization problems. Specifically, metaheuristics combine basic heuristic methods in higher level frameworks in order to find solutions to combinatorial problems at a reasonable computational cost [17, 18]. To guide the search, problem-specific *fitness function(s)* is defined, which helps to obtain good solutions from a potentially infinite search space.

Most of the real world testing problems include multiple objectives to be optimized at the same time, which often conflict among each other [19]. For instance, a test case prioritization problem requires dealing with multiple well-known contrasting criteria, such as maximizing statement coverage while minimizing the execution time of the test cases. SBST has shown to be widely effective in solving these problems in the literature [17, 20-24].

2.3 Search Algorithms

There exist different search algorithms that can be applied in SBST. Table 1 presents a classification of the selected search algorithms used in this thesis. In this section, we describe at high-level several types of search algorithms that are used in this thesis.

Genetic algorithms (GAs) are the most famous metaheuristic used in SBST. GAs are inspired by the process of natural selection that optimize one or more objectives (e.g., maximizing code coverage while minimizing the execution cost). GAs starts with a random population of solutions, where each individual represents a potential solution to the optimization problem. Each solution is evaluated using its *fitness*, and the population is evolved towards better solutions by generating new solutions using bio-inspired operators: *selection, crossover*, and *mutation* [25]. The *selection operator* selects candidate solutions within each generation, the *crossover operator* is used to recombine pairs of selected individuals, and *mutation operator* randomly modifies parts of individuals. For multi-objective optimization, weight-based genetic GA (WBGA) assigns a particular weight to

each objective function for converting a multi-objective problem into a single objective problem using a scalar function.

Table 1. Classification of the Selected Search Algorithms

Algorithm Category		Algorithm
Genetic Algorithms (GAs)	Weight-Based GA	WBGA
	Sorting-Based GA	NSGA-II
	Cellular-Based GA	MOCell
Evolutionary Algorithms (EAs)	Strength Pareto EA	SPEA2
	Evolution Strategies	PAES
	Indicator-Based	IBEA
Nearest Neighbor	Greedy	Greedy

Non-dominated Sorting Genetic Algorithm (NSGA-II) [10] is based on *Pareto optimality*. NSGA-II sorts the population and places them into several fronts based on the ordering of Pareto dominance. After that, the individual solutions are selected from the non-dominated fronts, and if the number of the solutions from the non-dominated front exceeds the specified population size, the solutions with a higher value of *crowding distance* are selected. Specifically, *crowding distance* is used to measure the distance between the individual solutions with the others in the population [26].

In Strength Pareto Evolutionary Algorithm (SPEA2), the fitness of each solution is calculated by adding up its raw fitness and density information [9]. The raw fitness is calculated based on the number of the solutions it dominates. The density information is calculated based on the distance between a solution and its nearest neighbors. SPEA2 starts by creating an empty archive and fills it with the non-dominated solution from the population, and in the subsequent generations, the solutions from the archive and the non-dominated solution in the current population are used to create a new population. Additionally, if the number of combined non-dominated solutions is more than the maximum size of the specified population, the solutions with the minimum distance to other solutions are selected by applying a truncation operator.

Indicator Based Evolutionary Algorithm (IBEA) allows any performance indicator (e.g., *hypervolume* (*HV*) [27]) to be incorporated into the selection mechanism of a multi-objective evolutionary algorithm [28]. Specifically, quality indicators are used to measure the quality of the solutions for multi-objective optimization, and they are used by IBEA to guide the search towards optimal solutions. For instance, *HV* calculates the volume in the objective space covered by a non-dominated set of solutions (e.g., *Pareto front*) [29]. One potential downfall of using *HV* is the computational complexity of calculating the hypervolume measure as the number of objectives increase.

Multi-Objective Cellular Genetic Algorithm (MOCell) is based on the cellular model of genetic algorithm with an assumption that an individual solution can only interact with its neighbors in the population during the search process [30]. Specifically, MOCell stores a set of non-dominated solutions in an external archive, and after each generation, MOCell replaces a fixed number of randomly chosen solutions in the population with the solutions from the archive with a feedback procedure until the termination criteria for the algorithm

are met. Note that the replacement only occurs if the solutions from the population are worse than the solutions in the archive.

Pareto Archived Evolution Strategy (PAES) maintain an archive of non-dominated solutions and applies the dynamic mutation operator for exploring the search space to find optimal solutions [31]. Initially, a random solution is added to the archive, which is then used to generate the offspring solution. If the newly generated solution is better than the parent solution, it is used to replace the parent solution and further added to the archive if it is better than the solutions there. On the other hand, if the generated solution is worse than the parent solution, it is discarded, and the parent solution is used to generate another solution. Additionally, if the generated solution is neither dominating nor dominated by the solutions in the archive, additional measures (similar to NSGA-II) are taken into account, i.e., if the generated solution lies in a more crowded part of the feasible place (with respect to members of the archive) as compared to the parent solution, it is disposed, else it replaces the parent solution.

The greedy algorithm works on the "next best" search principle, such that the element with the highest weight for the particular objective (e.g., statement coverage) is selected first [17]. After that, an element with the second highest weight, third highest weight and so on are selected on that order until all the elements are selected, or the termination criteria of the algorithm are met. If there exist multiple elements with the same weight, one of them is randomly selected [32]. For multi-objective optimization, greedy algorithm converts the problem into a single optimization problem using a scalar objective function. Specifically, each objective is assigned a weight (based on user preference), and the overall value of the objective is computed by summing up the weighted objective values.

2.4 Data Mining

Data mining is used to extract hidden correlations, patterns, and trends from the large data set, which are both understandable and useful to the data owner [33]. In our context, we used data mining to extract hidden correlations and patterns between the test cases based on their historical execution data for prioritizing the test cases dynamically. Data mining involves using pattern recognition technologies together with statistical and mathematical techniques [34]. Data mining techniques have been widely applied to problems from several domains (e.g., engineering), and it is one of the fastest growing fields in the computer industry [34].

Nowadays, a large amount of data is produced since all automated systems generate some form of data for analysis or diagnostic objectives [35]. However, the raw data might be unstructured and not immediately suitable for automated processing, and the data might be collected from different formats. To make better sense of the data, data mining consists of three phases: data cleaning, feature extraction, and algorithm design. Specifically, the unstructured and complex data is converted to a well-structured data set that can be effectively used by the computer program in the data cleaning and feature extraction phase. After that, an algorithm is used to discover hidden patterns from the data.

Data mining methods can be classified into supervised and unsupervised learning. Supervised learning is used for labeled data, and it aims to discover the relationship between input data (i.e., independent variables) and the target attribute (i.e., dependent variable) or outcome [36]. Unsupervised learning is used for unlabeled data, and it aims to identify hidden patterns within input data without labeled responses [36]. Moreover, supervised learning uses class information from the training instances, unlike unsupervised learning that does not use the class information. In our thesis, we have labeled data, and therefore, we adopted supervised learning. Supervised models have two main types: classification and regression models. Classification models map the input space into predefined classes, and regression models map the input space into a real-valued domain [36].

In this thesis, we used classification models since we have predefined classes. In classification models, the classification rules are constructed in two major ways: 1) Indirect method (e.g., C4.5 [37]), which learns decision trees and converts them to rules and 2) Direct method (e.g., Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [38]), which extracts rules from the data. The indirect method is computationally expensive in the presence of noisy data while the direct method has the over pruning (hasty generalization) problem [39]. To avoid these shortcomings, Pruning Rule-Based Classification (PART) [40] is derived from C4.5 and RIPPER. Specifically, PART creates partial trees and corresponding to each partial tree; a single is extracted for the branch that covers the maximum nodes.

3 Research Method

This section describes our research method used for the entire thesis. This research work was funded through Certus [41], which is a center for research-based innovation with the objective to improve the reliability of large-scale systems, and it involves some industrial partners. This thesis is mainly driven by the collaboration with Cisco Systems Norway, which focuses on testing VCSs in a cost-effective manner [7, 8].

3.1 Problem Identification

The thesis was initiated by identifying a research problem in an industrial setting (i.e., Cisco Systems Norway) to ensure practical relevance. The VCSs developed by Cisco enable high-quality conference meetings and can be configured in many ways for supporting various user requirements (e.g., using different communication protocols, such as SIP and H323) [42]. Testing VCSs in a cost-effective manner is a challenging task, and a large number of test cases have been designed and implemented by test engineers. Specifically, each test case needs to 1) setup test configurations for VCSs under test (i.e., SUTs), 2) invoke a set of test APIs, and 3) check the statuses of the VCSs after executing the test case as shown in Fig. 3. The test cases take a long time to execute (median execution time of a test case is 30 minutes), and it is not possible to execute all the test cases due to different constraints (e.g., time, resource). Based on the domain knowledge of VCS testing and discussions with test engineers, we observed that there are three challenges in their current testing practice that we aim to address on this PhD thesis.

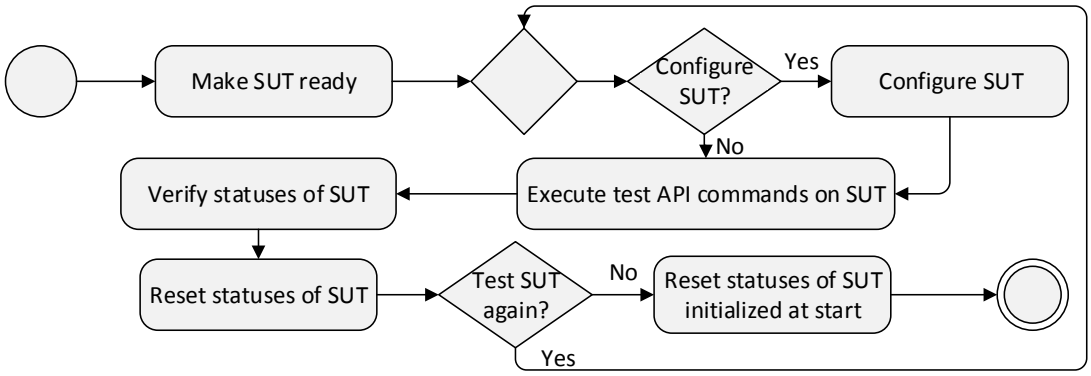

Fig. 3. Overview of testing a VCS (SUT)

Challenge 1. *Multi-Objective Optimization*. Our industrial partner (i.e., Cisco systems Norway) develops VCSs in a continuous integration environment, and 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. The median execution time of a test case is 30 minutes, and thus, it is not feasible to execute all the test cases due to a limited time budget (e.g., 10 hours available for test case execution). Therefore, it is important to seek cost-effective approaches to optimize a set of test cases based on predefined criteria, such as cover maximum number of configurations, test APIs, statuses, and detect faults as possible.

A number of existing works have been done in the context of Cisco for multi-objective test case optimization [7, 8, 43]. However, these works define the cost-effective objectives at a high level (e.g., feature pairwise coverage in [8], test resource usage in [7]) rather than taking detailed test configurations, test APIs, and statuses into account. Through further investigation, we noticed that these detailed levels of information should be properly incorporated when optimizing test cases before execution. Doing so makes it possible to propose efficient approaches that take configurations, test APIs, and statuses of the VCSs into account.

Challenge 2. *Test Case Implantation*. We noticed that the existing test cases in Cisco are unable to cover the entire configurations and test APIs supported by the VCSs. This may result in delivering VCSs (without thorough testing) to markets, and end users might encounter unexpected problems when using the VCSs in a way that is not tested. Moreover, we need to mention that developing test cases manually to cover the entire configurations and test APIs is practically infeasible since each configuration and test APIs can be configured with a number of values, e.g., configuration *protocol* can be *SIP* and *H323*, test API *dial* can be executed with a call rate between *64* and *6000*. Also, we observed that certain test cases verify similar configurations decreasing the efficiency of testing. Thus, it requires an automated and cost-efficient approach to modify the existing test cases to test maximum configurations and test APIs, which are not covered by the existing test cases.

Challenge 3. *Dynamic Test Case Prioritization*. Each time the software in VCSs is modified, a test cycle is performed, where a set of test cases from the test suite is executed. The result of the test execution data is then stored. With time, the amount of test execution data increases significantly. Currently, there exists a huge amount of historical test case execution data (more than 100 Gb) in the industrial partner, e.g., detailed configurations for execution, test APIs employed, success or failure after executing each test API, and the overall execution result of each test case at different test cycles. The test engineers at Cisco are interested to improve the testing process by incorporating the historical test case execution data in order to help them find bugs as soon as possible.

3.2 Problem Formulation

After identifying the challenges with our industry partner, we reviewed the state-of-the-art techniques to match the challenges [44-51]. Based on this, we proposed the following three research questions to address during my Ph.D.

RQ1. How to cost-effectively optimize a set of test cases based on predefined criteria? This research question aims to address challenge 1 by proposing cost-effective approaches to optimize a given set of test cases based on predefined criteria. Specifically, we focus on two perspectives for test case optimization: 1) *test case prioritization* that focuses on prioritizing test cases into an optimal order for maximizing the effectiveness of testing, and 2) *test case selection* that aims at cost-effectively selecting a set of relevant test cases

within a time budget for testing particular systems. Since we observed that executing the existing test cases cannot cover the entire configurations and test APIs (challenge 2), we require an efficient approach for modifying the existing test cases to test the uncovered configurations and test APIs, which motivates RQ2.

RQ2. How to cost-effectively implant existing test cases with the aim to test uncovered configurations and test APIs? This research question aims to address challenge 2 by modifying the existing test cases with the aim to test untested configurations and test APIs. We name this idea as *test case implantation*, i.e., implanting the untested information (e.g., configurations) into the existing test cases. Addressing RQ2 can largely help enhance the capability of existing test cases in terms of covering configurations and test APIs without a need to manually implement a large number of test cases. As test cases are executed, the size of the historical test case execution data increases, which can then be investigated to identify useful relations for refining the solutions from RQ1 and RQ2, which motivates RQ3.

RQ3. What relations can be mined and extracted from test cases based on their historical execution data in order to find faults as soon as possible? This research question aims to address challenge 3 by mining execution relations among test cases in order to find faults as soon as possible. Moreover, this research question can help refine the solutions from RQ1 and RQ2 (i.e., *multi-objective test case prioritization* and *test case implantation*).

In our work, we used search-based techniques since our potential solution space is huge and search-based techniques have shown good results for solving such complicated optimization problems. Moreover, we used different rule mining techniques that have shown good results in literature to extract execution results between the test cases in RQ3. In addition, we identified and formulated a set of objectives through careful investigation and discussion with test engineers. Note that the entire objectives were mathematically defined and formulated, and empirically evaluated after applying the search techniques.

3.3 Solution Realization

This step focuses on realizing the proposed solutions to address each research question (described in Section 3.2) for tacking the challenges described in Section 3.1. Table 2 presents a high-level overview of how different research questions were addressed in this PhD work. Initially, we discussed each challenge with Cisco to have a common understanding of the problem. After that, we obtained datasets from Cisco for tacking different challenges and discussed the characteristics of the datasets with the test engineers from Cisco to ensure that we understood it correctly. Based on this, we implemented our techniques and performed different empirical evaluations using the industrial case study. Once the results looked promising, we performed additional experiments with open source case studies to generalize the results. Finally, we discussed the findings with the engineers from Cisco (e.g., the benefits of the proposed techniques), and how the solution could be

integrated into their development process. For instance, for dynamic test case prioritization (Paper E), we discussed the discovered hidden rules among the execution relation of the test cases, and how it helped to discover the faults quicker. Additionally, we designed and implemented the corresponding tool support with respect to the realized solutions for each paper, e.g., STIPI to prioritize test cases (Paper A), search-based test case selection tool with user preferences (Paper B), cluster-based genetic algorithms (Paper C and Paper F), SBI to implant existing test cases (Paper D), and REMAP to dynamically prioritize the test cases (Paper E).

Table 2. Overview of Solutions for Addressing Different Research Questions and Challenges

Challenge	Research Question	Solution
Multi-Objective Optimization	1	Paper A, B, C
Test Case Implantation	2	Paper D
Dynamic Test Case Prioritization	3	Paper E

3.4 Solution Evaluation

To assess the performance of the proposed methods in terms of each research question, we used different case studies, such as industrial, real world, artificial problems, and open source. Moreover, we compared the proposed methods against the state-of-the-art approaches using widely used evaluation metrics, such as the average percentage of fault detected [52], statement coverage, branch coverage, mutation score [53], fault detected [54], and hypervolume [27, 55]. In addition, we applied rigorous statistical tests, such as Vargha and Delaney statistics [56] and Mann-Whitney U test [57] to analyze the results. For instance, to evaluate RQ3, we compared 18 variants of the proposed approach, REMAP (in order to find the best variant) against 29 variants of four approaches by employing two industrial case studies and three open source datasets using an average percentage of faults detected per cost [58] scores (Paper E). To statistically analyze the results (Paper E), we applied four different statistical tests: 1) Vargha and Delaney statistics, 2) Mann-Whitney U test, 3) Kruskal-Wallis test [59], and 4) Dunn's test [60] with Bonferroni correction [61]. The goal for such extensive evaluation is to ensure that the proposed approach can assist to improve the current testing practice and is robust.

4 Evolutionary Computation Based Testing Methods

In this section, we present a set of methods to address each research question.

4.1 Multi-Objective Test Optimization

4.1.1 Search-Based Test Case Prioritization Approach

Based on our collaboration with Cisco Systems Norway, the first step is to find an optimal method to cost-effectively optimize a set of test cases based on predefined criteria. Specifically, in the context of Cisco, test cases need to be prioritized by taking into account detailed test configurations, test APIs, and statuses of the systems since it is not possible to execute all the test cases due to different constraints (as described in Section 3.1). However, there exists trade-off among these objectives, and thus, the problem can be formulated as a multi-objective optimization problem.

To address this problem, we proposed a search-based test case prioritization approach based on *incremental unique coverage* and *position impact* (STIPI) in Paper A. Fig. 4 presents an overview of STIPI. Specifically, four effectiveness measures were defined with respect to the required objectives for prioritization based on the context of testing VCSs, i.e., maximizing configuration coverage (*CC*), test API coverage (*APIC*), status coverage (*APIC*), and fault detection capability (*FDC*). For each objective, fitness function was defined by incorporating two prioritization strategies: 1) *Incremental unique coverage*, i.e., for a specific test case, we only consider the incremental unique elements (e.g., test APIs) coverage by the test case as compared with the elements covered by the already prioritized test cases; and 2) *Position impact*, i.e., a test case with a higher execution position (i.e., scheduled to be executed earlier) has more impact on the quality of a prioritization solution. The fitness function can be combined with any multi-objective search algorithm (e.g., in our context, we used NSGA-II). The results were empirically evaluated using three datasets from Cisco with four different time budgets.

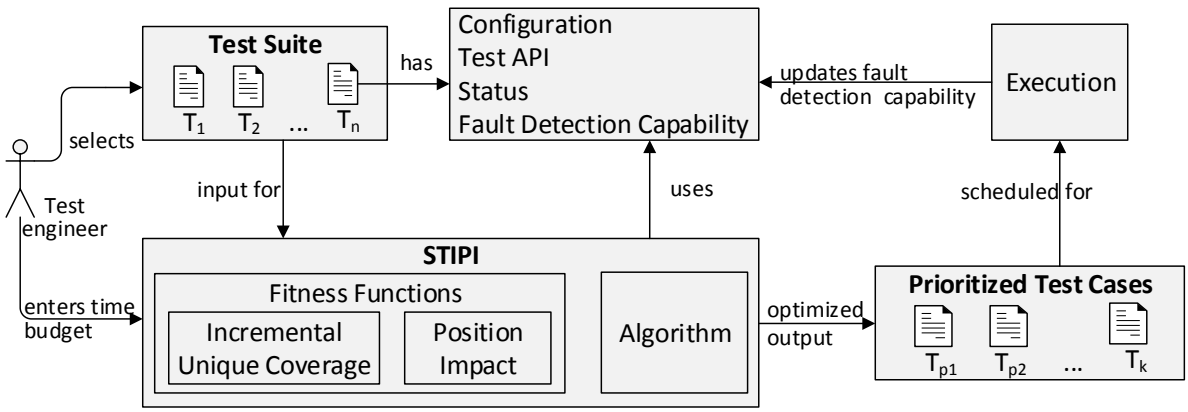

Fig. 4. Overview of STIPI

4.1.2 Search-Based Test Case Selection Approach

Besides Cisco, we also worked on a real-world case study from the maritime domain for testing some of the key elements of subsea oil and gas production systems. The case study was created using different standards (e.g., design and operation of subsea production systems- ISO 13628-6:200), OREDA offshore reliability data handbook, and requirements from different oil and gas companies publicly available. The real-world case study was inspired by our prior collaboration with an industrial partner from the maritime domain. The department collaborating with us used manual test cases, which requires a lot of effort (in terms of time and cost) to execute. However, they had a limited time budget available to test the systems, and thus, they needed to select the best set of test cases that could be executed within the time budget while ensuring maximum effectiveness.

To address the above-mentioned challenge, we introduced a search-based multi-objective test case selection approach (Fig. 5) in Paper B. Specifically, we defined four objectives: maximizing mean priority (*MPR*), mean probability (*MPO*), mean consequence (*MC*), and minimizing time difference (*TD*). For the selected test cases, 1) *MPR* measures the average importance of the test cases based on the type of requirements the test cases check; 2) *MPO* measures the average likelihood that the test cases might find faults; 3) *MC* measures the average impact of failures of the test cases that the system can have on the environment once it is operational; and 4) *TD* measures the difference between the execution time of the selected test cases and the time budget available for testing.

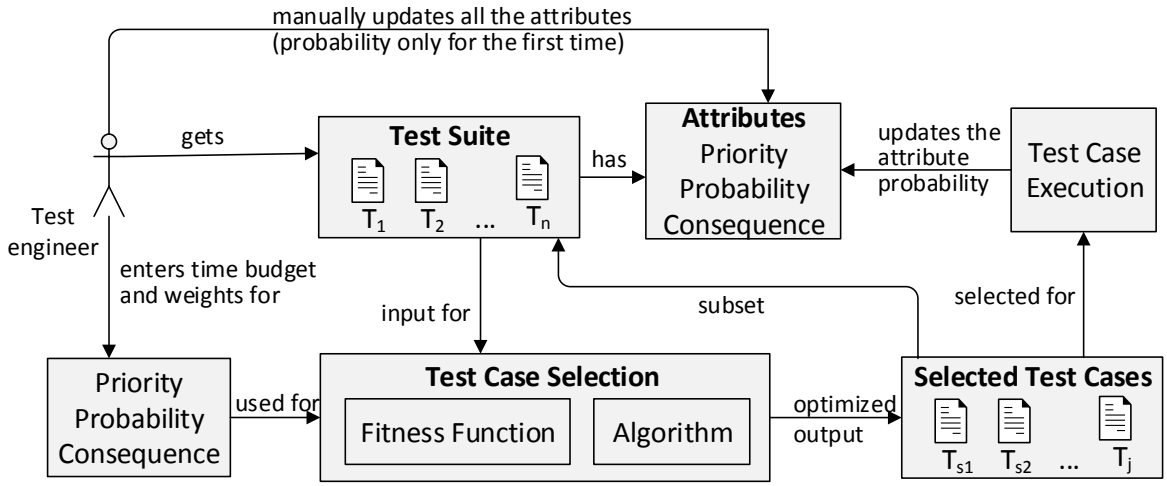

Fig. 5. Overview of search-based test case selection approach that supports user preferences

Moreover, to incorporate user preference for different cost-effectiveness measures, we proposed a fifth objective to guide Pareto-based search algorithms in a particular search space. The objectives were then used with eight different search algorithms (three weight-based and five Pareto-based) by employing a real-world case study and 10 artificial problems with varying size. We applied two different weight assigned strategies to incorporate user preferences: *fixed weights (FW)* and *randomly assigned weights (RAW)*. Specifically, *FW* assigns fixed normality weights to each objective based on the domain knowledge, while *RAW* is inspired by Random-Weighted Genetic Algorithm (RWGA),

where a randomly-generated set of normalized weights are dynamically assigned (to each objective) at each generation during the search [26]. Note that the weights generated by *RAW* should satisfy user-defined constraints. Overall 75 problems were defined for the real world case study, and 750 problems were defined for the 10 artificial problems based on possible user preferences.

4.1.3 New Cluster-Based Genetic Algorithms to Support Multi-Objective Test Optimization

While working on the multi-objective test case optimization problems (Paper A and B), we observed that the existing multi-objective search algorithms make choices based on the random number generation when selecting parent solutions (i.e., stochastic parent selection) to produce offspring solutions, due to the selection mechanisms employed in the algorithms. For example, in binary tournament selection (commonly used in the literature [62, 63]), two solutions are randomly selected, and the better solution is selected as the parent. However, if the selected parent solutions are suboptimal in the population, it might result in offspring solutions with bad quality (i.e., the produced offspring solutions have worse value for the objectives as compared to the solutions in the populations). This may subsequently degrade the overall quality of the solutions in the next generation, and in the worst case, stochastic parent selection may prevent algorithms in finding optimal solutions.

To address this problem, we proposed a cluster-based genetic algorithm with elitist selection (CBGA-ES – Paper F) and its extension CBGA-ES$^+$ (Paper C) for supporting multi-objective test optimization. The core-idea of CBGA-ES and CBGA-ES$^+$ lies on 1) dividing the population into different clusters for grouping solutions with similar quality and 2) defining *cluster dominance strategy* to rank the different clusters and only choosing the solutions from the best clusters for producing offspring solutions. When a new population is created, this process will be repeated for producing the next generation until the termination conditions for the algorithm is met. The core difference between CBGA-ES and CBGA-ES$^+$ lies on how the solutions from the different clusters are selected. Specifically, CBGA-ES$^+$ selects only non-dominated solutions from the best clusters, while CBGA-ES selects all the solutions from the best clusters, and in case the number of the solutions in the cluster is higher than the specified elite population size, random solutions are selected.

To empirically evaluate CBGA-ES and CBGA-ES$^+$, we employed multiple case studies for five different multi-objective test optimization problems (e.g., test case prioritization problem (Paper A), test case selection (Paper B)) using three industrial and five open source case studies. The other test optimization problems were borrowed from a well-known Search-Based Software Engineering Repository hosted by the CREST center [64]. Finally, the results were compared with the state-of-the-art search algorithms.

4.2 Search-Based Test Case Implantation Approach

Followed by the multi-objective test case optimization step, we focused on how to cost-effectively test the untested configurations and test APIs in Cisco to address RQ2. Manually implementing test cases (e.g., specifying configurations, calling relevant test API commands, checking corresponding system status) to test configurations and test API commands require a large amount of manual work, which is practically infeasible. Moreover, we noticed that configuring each test case is expensive since the VCSs need to be reset and reconfigured according to the specific test case requirements and some test cases verify similar configurations, which can decrease the efficiency of testing.

To address the above-mentioned challenge, we proposed a search-based test case implantation approach (named as SBI) in Paper D to automatically analyze and implant the existing test suite cost-effectively with the aim to test the untested configurations and test API commands. SBI consists of two key components (Fig. 6): *test case analyzer* and *test case implanter*. The *test case analyzer* component ensures that implanted test cases are semantically correct (e.g., two new statements need to be added in a particular order). For this, the *test case analyzer* component statically analyzes each test case in the original test suite to obtain the program dependence graph [65, 66] for each *test method* to obtain the dependencies among test case statements. After that, the *test case implanter* component uses multi-objective search to select suitable test case for implantation, and the selected test cases are modified using a mutation operator at the test case level using three operations: *addition*, *modification*, and *selection*. Specifically, changes are made to one or more test case statements using the program dependence graph obtained from the *test case analyzer* component as shown in Fig. 6.

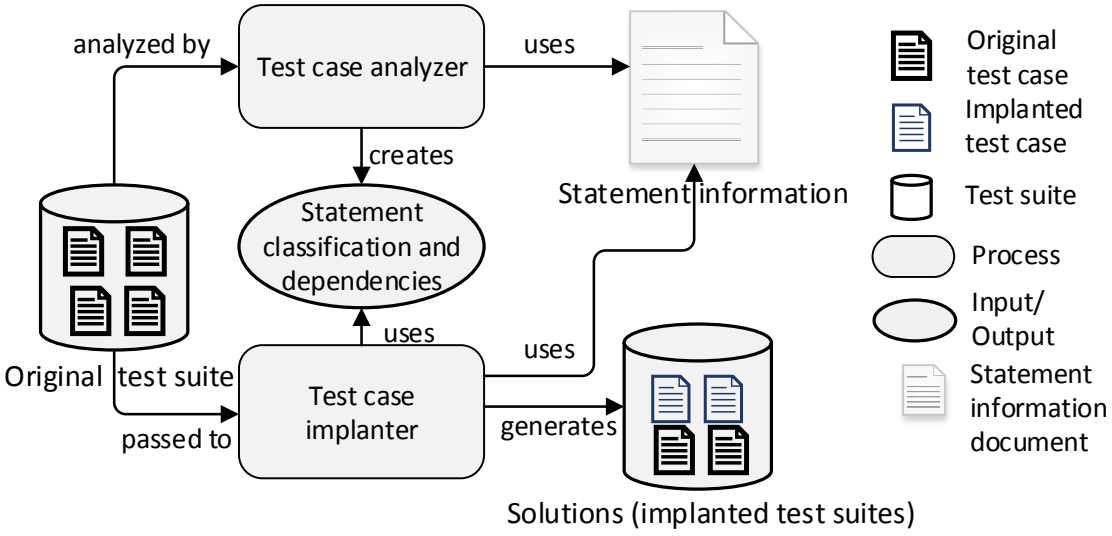

Fig. 6. Overview of SBI

Five objectives were identified and defined to guide the search: maximizing number of configuration variable values covered and pairwise coverage of parameter values of test API commands, and minimizing number of implanted test cases, number of changed

statements, and estimated execution time of the test cases. The proposed approach was empirically evaluated with an industrial and an open source case study using three variants of SBI (using NSGA-II, weight-based genetic algorithm, and random search).

4.3 Rule-Mining and Search-Based Dynamic Test Case Prioritization Approach

There exists a huge amount of historical test case execution data in the industrial partner, which grows in size as more test cases are executed. Understanding the relationships among the test case execution results can help to find faults quicker. In this regard, we carefully investigated the historical test case execution data and observed that there exist underlying relations among the executions of test cases, which test engineers were not aware of when developing the test cases. For instance, when the test case (*T1*) that tests the amount of free memory left in VCS after pair to pair communication for a certain (e.g., 5 minutes) *fails*, the test case (*T2*) verifying that the speed of fan in VCS locks near the speed set by the user always *fails* as well. In addition, we observed that when *T2* is executed as *pass*, another test case (*T3*) that checks the CPU load measurement of VCS also always *passes*. We observed this relation in spite of the fact that all test cases are supposed to be executed independently of one another.

Mining such execution relations among test cases can improve the effectiveness of test case prioritization. For instance, if *T1* is executed as fail, *T2* should be executed as early as possible (e.g., right after executing *T1*) since *T2* has a high chance to fail, i.e., detect a fault. In addition, if *T2* is executed as *pass*, *T3* should be executed later (i.e., the priority of *T3* should be decreased). Therefore, it is essential to prioritize the test cases in a dynamic manner considering the runtime test case execution results.

With such motivation in mind, we proposed a test case prioritization approach, REMAP (Paper G, and its extension Paper E), which uses rule mining and multi-objective search to prioritize the test cases. As shown in Fig. 7, 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 (e.g., *RIPPER*). Second, *SP* defines multiple objectives (e.g., fault detection capability) to statically prioritize the test cases using a multi-objective search algorithm (e.g., NSGA-II). Third, *DEP* executes the statically prioritized test cases obtained from the *SP* and dynamically updates the order of the unexecuted test cases based on the runtime test case execution results using the *fail* and *pass rules* from *RM*.

We empirically evaluated 18 variants of REMAP using three rule mining algorithms (RIPPER, C4.5, and PART) and three muti-objective search algorithms (NSGA-II, IBEA, and SPEA2) using two different sets of objectives (two objectives and three objectives). The 18 variants of REMAP were compared against two variants of random seach, three variants of greedy, 18 variants of static search-based prioritization approaches, and six

variants of rule-based approaches by employing two industrial case studies and three open source case studies.

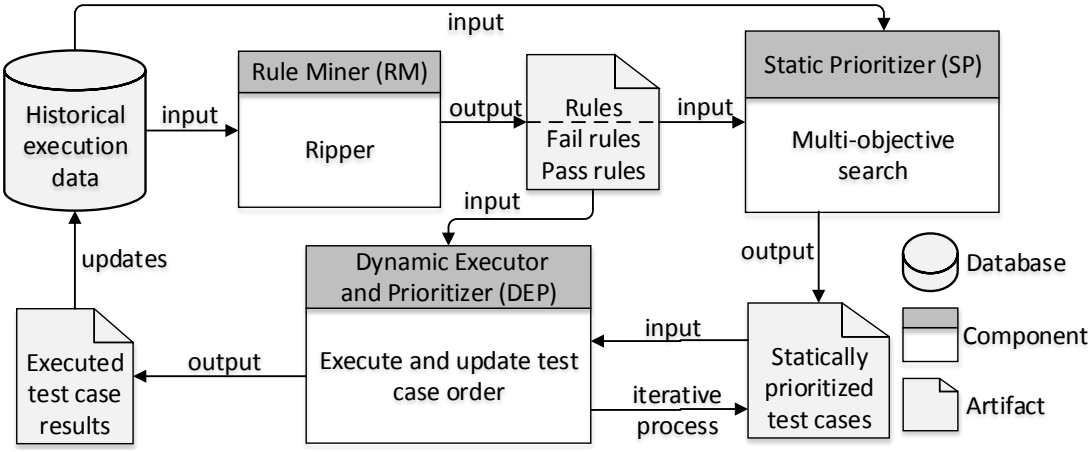

Fig. 7. Overview of REMAP

5 Summary of Results

In this section, a summary and key results are presented for each paper submitted as a part of this thesis.

5.1 Paper A

"STIPI: Using Search to Prioritize Test Cases Based on Multi-Objectives Derived from Industrial Practice", D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen. In: Proceedings of the 28th International Conference on Testing Software and Systems (ICTSS), pp. 172-190. Springer, 2016. DOI: 10.1007/978-3-319-47443-4_11.

This paper aims at tackling the test case prioritization problem. Specifically, test case prioritization is one of the most widely used test optimization approach with the aim to prioritize a set of test cases into an optimal order for achieving certain criteria (e.g., fault detection) as early as possible. While working on a research-based innovation project with Cisco to improve the reliability of Video Conferencing systems (VCSs), we noticed that testing VCSs there is a challenging task, and a large number of test cases have been designed and implemented, which take a lot of time to execute (e.g., the median execution time of a test case is 30 minutes). To address this challenge, we defined a multi-objective test case prioritization problem based on the context of testing VCSs. Specifically, the proposed approach named *Search-based Test case prioritization based on Incremental unique coverage and Position Impact (*STIPI*)* defines four objectives and incorporates two prioritization strategies to evaluate the quality of the prioritized solutions. After that, the fitness functions are incorporated with a multi-objective search algorithm.

We evaluated STIPI (using NSGA-II) against random search (RS), greedy approach, and three approaches from the existing literature by employing three datasets from Cisco with four different time budgets: 25%, 50%, 75%, and 100%. Note that 100% time budget implies that time budget is equal to the overall execution time of the test suite. Therefore, in total 60 comparisons were made with STIPI (i.e., 5 approaches × 4 time budgets × 3 datasets). The results showed that STIPI significantly outperformed the selected approaches for 90% of the comparisons (i.e., 54 out of 60). Moreover, STIPI achieved better performance than RS for on average 39.9%, 18.6%, 32.7%, and 43.9% in terms of *average percentage of configuration coverage*, *average percentage of test API coverage*, *average percentage of status coverage*, and *measured fault detection capability*, respectively.

5.2 Paper B

"Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study", D. Pradhan, S. Wang, S. Ali, and T. Yue. In: Proceedings of Genetic and Evolutionary Computation Conference (GECCO), pp. 1085-1092. ACM, 2016. DOI: DOI: 10.1145/2908812.2908850.

This paper focuses on the test case selection problem. Specifically, the problem focuses on selecting a subset of test cases from the existing test suite, where the order of the selected test cases does not matter. In some contexts (e.g., releasing the software to the customer at the data defined in the contract) it is known beforehand the time budget (i.e., duration) available for testing. However, it is often infeasible to execute all the test cases within the given time budget, and the test manager/test engineers need to select the optimal set of test cases that can be executed within the time budget. To address this challenge, we proposed a search-based multi-objective test case selection approach, where the test engineers can provide a preference for different objectives, e.g., give more weight to the test cases, which found faults in the earlier executions.

To evaluate our approach, we performed an empirical study using 1) three different weight-based search algorithms: Alternating Variable Method (AVM) [67], GA, and (1+1) Evolutionary Algorithm (EA) [68], and 2) five different Pareto-based search algorithms: NSGA-II, MOCell, SPEA2, CellDE, and IBEA by employing a real-world case study and 10 artificial problems with varying size. We created artificial problems by simulating the real-world case study. For each approach, two different weight assignment strategies: 1) *Fixed weights* and 2) *Randomly assigned weights* were applied, and RS was used for a sanity check. The results showed that all the search algorithms performed significantly better than RS, and SPEA2 with either of the weight assignment strategies performed the best among the search algorithms. Overall, SPEA2 managed to improve on average 32.7%, 39%, and 33% in terms of *MPR*, *MPO*, and *MC* as compared to RS.

5.3 Paper C

"CBGA-ES^{+}: A Cluster-Based Genetic Algorithm with Non-Dominated Elitist Selection for Supporting Multi-Objective Test Optimization", D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen. Published in IEEE Transactions on Software Engineering (TSE). DOI: 10.1109/TSE.2018.2882176.

Existing multi-objective search algorithms make choices based on random number generation when selecting parent solutions to produce offspring solutions (i.e., stochastic parent selection), due to the selection mechanisms employed in the algorithms. However, if the selected parent solutions are suboptimal in the population, it might result in offspring solutions with bad quality. This may subsequently degrade the overall quality of the

solutions in the next generation, and in the worst case, stochastic parent selection may prevent algorithms in finding optimal solutions. To address this issue, we proposed a cluster-based genetic algorithm with elitist selection (CBGA-ES) for supporting multi-objective test optimization in Paper F.

In this paper, we extended the CBGA-ES algorithm to select only the non-dominated solutions from different clusters to produce offspring solutions, CBGA-ES$^+$. In addition, we extensively evaluated the performance of CBGA-ES$^+$ by comparing it with predecessor CBGA-ES, RS, Greedy, and four selected search algorithms (i.e., NSGA-II, SPEA2, MOCell, and PAES) that managed to obtain better performance in literature. For the empirical evaluation, we employed five multi-objective test optimization problems: test case prioritization (*TCP*), test case selection (*TCS*), test suite minimization (*TSM*), testing resource allocation (*TRA*), and integration and test order (*ITO*). *TSM* problem aims to eliminate the redundant test cases from the existing test suite for the current system in order to reduce the cost of testing. *TRA* problem aims to optimally allocate the resources to different modules (that comprise the software system) for maximizing the reliability while minimizing the test resources (e.g., cost) [23]. *ITO* problem focuses on determining an order to integrate and test the units (e.g., classes) to minimize the stubbing cost, where a stub is an emulation of a unit that has not yet been implemented or integrated into the software [69].

We used eight datasets (two industrial, one real world, and five open source) for the five multi-objective test optimization problems for a total of 20 experiments (i.e., *TSM*, *TCP*, and *TCS* × 6 datasets + 2 datasets for *TRA* and *ITO*). All the algorithms were compared with their best settings as obtained using the iRace optimization package [70]. To evaluate the results, we applied different evaluation metrics: *hypervolume (HV)*, *generational distance (GD)*, *generated spread (GS), fault detection (FD),* and the *average percentage of fault detected (APFD)*. The results showed that CBGA-ES$^+$ managed to significantly outperform 1) RS for 100%, 2) Greedy for 85%, and 3) CBGA-ES and the four selected search algorithms for an average of 66% of the experiments (in terms of the overall quality as indicated by *HV*). Moreover, for solutions in the same search space, CBGA-ES$^+$ managed to perform better than CBGA-ES and the four selected algorithms for an average of 11.5% of the objectives. In addition, CBGA-ES$^+$ achieved a better fault detection of more than 10% on average as compared to the selected search algorithms.

5.4 Paper D

"Automated Test Case Implantation to Test Untested Configurations: A Cost-Effective Search-Based Approach", D. Pradhan, S. Wang, T. Yue, S. Ali, and M. Liaaen. Revision submitted to Journal of Information and Software Technology (IST), Elsevier.

Large-scale systems are highly configurable allowing the users to run the systems with many different configurations. Thus, the test engineers need to implement a large number of test cases, and even then, many configurations remain untested. Manually writing test cases to test those untested configurations require a large amount of manual work, which is practically infeasible. Moreover, certain test cases verify similar configurations decreasing the efficiency of testing.

In this paper, we proposed a search-based test case implantation approach (SBI) to automatically implant an existing test suite with the aim to 1) achieve a higher coverage of configuration variable values, 2) cover more combinations of parameter values of test API commands, and 3) increase the efficiency of testing by modifying or removing redundant *test methods* that cover same configuration variable values or parameter values of test API commands. SBI includes two key components: 1) *Test case analyzer* that statically analyzes each test case to obtain the program dependence graph for the test statements; and 2) *Test case implanter* that uses multi-objective search to select suitable test case for implantation using three operators: *selection*, *crossover*, and *mutation*, and implants the selected test cases using a *mutation operator* at the test case level using three operations: *addition*, *modification*, and *deletion*.

Three variants of SBI (using NSGA-II, weight-based genetic algorithm, and random search) were evaluated with each other and against the original test suite using one case study from Cisco and an open source case study. All the algorithms were compared in their best settings obtained using the iRace optimization package. The results showed that all the three variants of SBI managed to significantly increase the effectiveness of the original test suite without significantly increasing the cost. Among the three variants of SBI, SBI with NSGA-II performed the best. Specifically, SBI with NSGA-II achieved on average 19.3% higher number of configuration variable values and 57.0% higher pairwise coverage of parameter values of test API commands for the two datasets. Additionally, for the open source case study, SBI with NSGA-II managed to improve statement coverage (*SC*), branch coverage (*BC*), and mutation score (*MS*) with on average 5.0%, 7.9%, and 3.2%, respectively. Note that we cannot apply *SC*, *BC*, and *MS* to the industrial case study since we do not have access to the source code.

5.5 Paper E

"Employing Rule Mining and Multi-Objective Search for Dynamic Test Case Prioritization", D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen. Submitted to Journal of Systems and Software (JSS), Elsevier.

In practice, regression test cases are executed each time the source code of the system is changed. Therefore, as time passes, there is a huge amount of historical test case execution data. This data can be used to further refine the prioritized solutions in order to detect the fault as soon as possible. For instance, we noticed that in the context of Cisco there exists a

hidden execution result relation among the test cases. Based on this observation, we introduced a test case prioritization approach, REMAP in paper G, which uses rule mining and multi-objective search to dynamically prioritize the test cases.

REMAP consists of three key components: *Rule Miner* (*RM*), *Static Prioritizer* (*SP*), and (*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 one of the most widely used rule-mining algorithms, *RIPPER*. Second, *SP* defines two objectives to statically prioritize the test cases using the algorithm NSGA-II. Third, *DEP* executes the statically prioritized test cases obtained from the *SP* and dynamically updates the test case order based on the runtime test case execution results using the *fail* and *pass rules* from *RM*.

In this paper (Paper E), we conducted an extensive empirical evaluation of REMAP by employing three different rule mining algorithms and three different multi-objective search algorithms. Moreover, we evaluated REMAP with one additional objective (i.e., execution time) for a total of 18 different configurations (i.e., $3\ rule\ mining\ algorithms \times 3\ search\ algorithms \times 2\ different\ set\ of\ objectives$) of REMAP. Specifically, we used three classification models: RIPPER, C4.5, and PART, and three representative multi-objective search algorithms: NSGA-II, SPEA2, and IBEA. The 18 different configurations of RIPPER were compared with 1) two variants of RS, 2) three variants of Greedy, 3) 18 variants of search-based approach (SBTP), and 4) six variants of rule-based prioritization approach (RBP) by employing five case studies (i.e., two industrial and five open source).

The results showed that the test cases prioritized by all the 18 variants of REMAP performed significantly better than the two variants of RS for all the case studies. Also, the two best variants of REMAP with two objectives and three objectives (i.e., REMAP with RIPPER and SPEA2 with two objectives and REMAP with RIPPER and IBEA with 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 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 average percentage of faults detected per cost ($APFD_c$) scores, respectively for the five case studies.

6 Future Directions

This section discusses possible future directions based on the above-mentioned four testing problems: test case prioritization, test case selection, test case implantation, and dynamic test case prioritization together with the proposed cluster-based genetic algorithms for supporting multi-objective test optimization.

For test case prioritization, the possible future direction is to compare our search-based test case prioritization approach based on incremental unique coverage and position impact with more state-of-the-art prioritization approaches using additional case studies with the larger scale to further generalize the results.

Concerning test case selection, the short-term plan is to apply different evaluation metrics (e.g., generational distance [71]) to evaluate the different search algorithms, and provide a general recommendation of when to use which search algorithm for multi-objective test case selection. Another possible future direction is to hybridize evolutionary algorithms with local search algorithms to evaluate whether a better performance can be achieved for our test case selection problem.

Concerning test case implantation, we aim to conduct additional experiments to study the impact of the proposed fitness functions in configuration coverage, statement coverage, branch coverage, and mutation score. Moreover, the long-term plan is to employ more case studies to further strengthen the application of the proposed search based implantation approach.

With regards to dynamic test case prioritization, 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. Additionally, we aim to conduct experiments with more case studies to further generalize the results.

Regarding the proposed cluster-based genetic algorithms to support multi-objective test optimization, the first plan is to involve industrial practitioners to deploy and assess CBGA-ES and CBGA-ES$^+$ in real industrial settings. The second plan is to study the impact of fitness evaluations (i.e., termination criteria) on the performance of the two algorithms. The third plan is to apply CBGA-ES and CBGA-ES$^+$ for multi-objective software engineering optimization problems from different domains to further strengthen the two algorithms.

7 Conclusion

This thesis proposed a set of methods based on evolutionary computation for cost-effective testing of large-scale systems. Driven by an industrial need, the methods aim to address four main problems: 1) *test case prioritization*, 2) *test case selection*, 3) *test case implantation*, and 4) *dynamic test case prioritization*. For each problem, we discussed the proposed method, and how it can address it. Moreover, we presented thorough empirical evaluation by comparing the proposed approach with the state-of-the-art techniques by employing various case studies (e.g., industrial, open source, real-world). Also, we provided tool support for each solution.

Specifically, for *test case prioritization* (Paper A), we proposed a search-based test case prioritization approach based on incremental unique coverage and position impact (*STIPI*). STIPI prioritizes test cases by taking into account four objectives: maximizing configuration coverage (*CC*), test API coverage (*APIC*), status coverage (*APIC*), and fault detection capability (*FDC*). These objectives were defined based on the context of testing VCSs. The results showed that *STIPI* managed to significantly outperform the existing approaches, and thus it can help test engineers to prioritize the test cases effectively.

For *test case selection* within a time budget (Paper B), we introduced a search-based multi-objective test case selection approach, which can incorporate user preference for different objectives. We defined four objectives to guide the search: maximizing mean priority (*MPR*), mean probability (*MPO*), mean consequence (*MC*), and minimizing time difference (*TD*). Also, an additional objective was introduced to incorporate user preference for different objectives for Pareto-based search algorithms. The results showed that search algorithms managed to perform significantly better than random search, and among the different search algorithms, SPEA2 managed to perform the best. Thus, it can be inferred from the results that it might be beneficial to use SPEA2 for similar test case selection problems.

Regarding *test case implantation* (Paper D), we proposed a search-based test case implantation approach (named as SBI). Specifically, SBI includes two key components: 1) *Test case analyzer* that statically analyzes each test case in the original test suite to obtain the program dependence graph for the test statements; and 2) *Test case implanter* that uses multi-objective search to select suitable test cases for implantation and implants the selected test suites using a mutation operator at the test case level. SBI can automatically analyze and implant the test suite cost-effectively to test the untested configurations. The results of the evaluation showed that SBI significantly managed to significantly increase the effectiveness of the original test suite without significantly increasing the cost. This can help test engineers to improve the quality of the test cases and automatically cover more configurations automatically, which can improve the effectiveness of testing.

For *dynamic test case prioritization* (Paper E), we proposed a rule mining and multi-objective search-based test case prioritization approach (REMAP). REMAP consists of three key components: 1) *Rule miner* that defines *fail rules* and *pass rules* for representing

the execution relations among test cases and mines these rules from the historical execution data; 2) *Static prioritizer* that defines objectives and applies multi-objective search to statically prioritize the test cases; and 3) *Dynamic executor and prioritizer* that executes the statically prioritized test cases obtained from the *static prioritizer* and dynamically updates the test cases order based on the runtime test case execution results and the mined rules from *rule miner*. The results showed that REMAP managed to significantly improve the results as compared to the state-of-the-art test case prioritization techniques. Moreover, test cases prioritized using REMAP had better *APFD* and $APFD_c$ values, which implies that REMAP has better fault detection capabilities.

In addition, the thesis proposed *cluster-based genetic algorithms* (Paper C and F) to support multi-objective test optimization problems. The core idea of these algorithms is to: 1) divide the population into different clusters for grouping solutions with similar qualities; and 2) define cluster dominance strategy to rank the different clusters and only select the solutions from the best clusters for producing offspring solutions. These algorithms were compared with the state-of-the-art by employing multiple case studies (e.g., industrial, open source) for five different multi-objective test optimization problems (e.g., test case prioritization). The results showed that the proposed algorithms managed to significantly outperform the state-of-the-art search algorithms for these test optimization problems, which show the benefit of using them for similar problems.

References for the Summary

[1] P. Ammann and J. Offutt, *Introduction to software testing*: Cambridge University Press, 2016.

[2] T. Shepard, M. Lamb, and D. Kelly, "More testing should be taught," *Communications of the ACM,* vol. 44, no. 6, pp. 103-108. 2001.

[3] L. Luo, "Software testing techniques," *Institute for software research international Carnegie mellon university Pittsburgh, PA,* vol. 15232, no. 1-19, p. 19. 2001.

[4] N. S. Eickelmann and D. J. Richardson, "What makes one software architecture more testable than another?," in *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pp. 65-67. ACM, 1996.

[5] D. Ganesan, J. Knodel, R. Kolb, U. Haury, and G. Meier, "Comparing costs and benefits of different test strategies for a software product line: A study from testo ag," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pp. 74-83. IEEE, 2007.

[6] J. Tretmans, "Model based testing with labelled transition systems," in *Formal methods and testing*, ed: Springer, 2008, pp. 1-38.

[7] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "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*, pp. 182-191. ACM, 2016.

[8] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software,* vol. 103, pp. 370-391. Elsevier, 2015.

[9] E. Zitzler, M. Laumanns, and L. Thiele, "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), 2001.

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* vol. 6, no. 2, pp. 182-197. IEEE, 2002.

[11] J. D. Arthur, M. K. Groner, K. J. Hayhurst, and C. M. Holloway, "Evaluating the effectiveness of independent verification and validation," *Computer,* vol. 32, no. 10, pp. 79-83. 1999.

[12] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology,* vol. 43, no. 14, pp. 833-839. 2001.

[13] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary Computation,* vol. 8, no. 2, pp. 173-195. MIT Press, 2000.

[14] A. S. Sayyad and H. Ammar, "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*, pp. 21-27. IEEE, 2013.

[15] N. Srinvas and K. Deb, "Multi-objective function optimization using non-dominated sorting genetic algorithms," *Evolutionary Computation,* vol. 2, no. 3, pp. 221-248. MIT Press, 1994.

[16] C. A. C. Coello, D. A. Van Veldhuizen, and G. B. Lamont, *Evolutionary algorithms for solving multi-objective problems* vol. 242: Springer, 2002.

[17] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering,* vol. 33, no. 4, pp. 225-237. IEEE, 2007.

[18] C. R. Reeves, *Modern heuristic techniques for combinatorial problems*: John Wiley & Sons, Inc., 1993.

[19] M. Harman, "Making the case for MORTO: Multi objective regression test optimization," in *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 111-114. IEEE, 2011.

[20] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software,* vol. 83, no. 4, pp. 689-701. Elsevier, 2010.

[21] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 140-150. ACM, 2007.

[22] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed, "Multi-objective test case prioritization in highly configurable systems: A case study," *Journal of Systems and Software,* vol. 122, pp. 287-310. 2016.

[23] Z. Wang, K. Tang, and X. Yao, "Multi-objective approaches to optimal testing resource allocation in modular software systems," *IEEE Transactions on Reliability,* vol. 59, no. 3, pp. 563-575. IEEE, 2010.

[24] G. Guizzo, S. R. Vergilio, A. T. Pozo, and G. M. Fritsche, "A multi-objective and evolutionary hyper-heuristic applied to the Integration and Test Order Problem," *Applied Soft Computing,* vol. 56, pp. 331-344. Elsevier, 2017.

[25] J. Brownlee, *Clever algorithms: nature-inspired programming recipes*: Lulu, 2011.

[26] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety,* vol. 91, no. 9, pp. 992-1007. Elsevier, 2006.

[27] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach," *IEEE transactions on Evolutionary Computation,* vol. 3, no. 4, pp. 257-271. 1999.

[28] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *International Conference on Parallel Problem Solving from Nature*, pp. 832-842. Springer, 2004.

[29] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham, "AbYSS: Adapting scatter search to multiobjective optimization," *IEEE Transactions on Evolutionary Computation,* vol. 12, no. 4, pp. 439-457. IEEE, 2008.

[30] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "Mocell: A cellular genetic algorithm for multiobjective optimization," *International Journal of Intelligent Systems,* vol. 24, no. 7, pp. 726-746. Wiley Online Library, 2009.

[31] J. Knowles and D. Corne, "The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation," in *Proceedings of the Congress on Evolutionary Computation*. IEEE, 1999.

[32] S. Li, N. Bian, Z. Chen, D. You, and Y. He, "A simulation study on some search algorithms for regression test case prioritization," in *International Conference on Quality Software (QSIC)*, pp. 72-81. IEEE, 2010.

[33] D. J. Hand, "Principles of data mining," *Drug safety,* vol. 30, no. 7, pp. 621-622. 2007.

[34] D. T. Larose, *Introduction to data mining*: Wiley Online Library, 2005.

[35] C. C. Aggarwal, *Data mining: the textbook*: Springer, 2015.

[36] O. Maimon and L. Rokach, "Introduction to knowledge discovery and data mining," in *Data Mining and Knowledge Discovery Handbook*, ed: Springer, 2009, pp. 1-15.

[37] J. R. Quinlan, "C4. 5: Programming for machine learning," *Morgan Kauffmann,* vol. 38, p. 48. 1993.

[38] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 115-123. 1995.

[39] G. Holmes, M. Hall, and E. Prank, "Generating rule sets from model trees," in *Australasian Joint Conference on Artificial Intelligence*, pp. 1-12. Springer, 1999.

[40] E. Frank and I. H. Witten, "Generating accurate rule sets without global optimization," 1998.

[41] *Certus*. Available: http://certus-sfi.no/

[42] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "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*, pp. 172-190. Springer, 2016.

[43] S. Wang, S. Ali, A. Gotlieb, and M. Liaaen, "A systematic test case selection methodology for product lines: results and insights from an industrial case study," *Empirical Software Engineering,* pp. 1-37. 2014.

[44] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability,* vol. 22, no. 2, pp. 67-120. Wiley Online Library, 2012.

[45] Q. Gu, B. Tang, and D. Chen, "Optimal regression testing based on selective coverage of test requirements," in *International Symposium on Parallel and Distributed Processing with Applications*, pp. 419-426. IEEE, 2010.

[46] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 129-139. ACM, 2007.

[47] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Information and Software Technology,* vol. 48, no. 7, pp. 586-605. 2006.

[48] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 218-227. IEEE Computer Society, 2008.

[49] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 257-266. ACM, 2010.

[50] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard, "Automatic error elimination by multi-application code transfer," 2014.

[51] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 257-269. ACM, 2015.

[52] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 179-188. IEEE, 1999.

[53] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th international conference on Software Engineering*, pp. 100-107. IEEE Computer Society Press, 1993.

[54] L. Vidács, Á. Beszédes, D. Tengeri, I. Siket, and T. Gyimóthy, "Test suite reduction for fault detection and localization: A combined approach," in *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014*, pp. 204-213. IEEE, 2014.

[55] J. Knowles, L. Thiele, and E. Zitzler, "A tutorial on the performance assessment of stochastic multiobjective optimizers," *Tik report,* vol. 214, pp. 327-332. ETH Zurich, 2006.

[56] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132. 2000.

[57] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics,* pp. 50-60. JSTOR, 1947.

[58] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 329-338. IEEE Computer Society, 2001.

[59] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association,* vol. 47, no. 260, pp. 583-621. 1952.

[60] O. J. Dunn, "Multiple comparisons using rank sums," *Technometrics,* vol. 6, no. 3, pp. 241-252. 1964.

[61] C. Bonferroni, "Teoria statistica delle classi e calcolo delle probabilita," *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze,* vol. 8, pp. 3-62. 1936.

[62] A. Shukla, H. M. Pandey, and D. Mehrotra, "Comparative review of selection techniques in genetic algorithm," in *International Conference on Futuristic Trends on Computational Analysis and Knowledge Management*, pp. 515-519. IEEE, 2015.

[63] A. Brindle, "Genetic algorithms for function optimization," Doctoral dissertaion and Technical Report TR81-2, University of Alberta, Department of Computer Science, Edmonton, 1981.

[64] Y. Zhang, M. Harman, and A. Mansouri, "The SBSE repository: A repository and analysis of authors and research articles on search based software engineering," *crestweb. cs. ucl. ac. uk/resources/sbse repository,* 2012.

[65] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *ACM Sigplan Notices*, pp. 177-184. ACM, 1984.

[66] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 9, no. 3, pp. 319-349. 1987.

[67] B. Korel, "Automated software test data generation," *Software Engineering, IEEE Transactions on,* vol. 16, no. 8, pp. 870-879. 1990.

[68] S. Droste, T. Jansen, and I. Wegener, "On the analysis of the (1+ 1) evolutionary algorithm," *Theoretical Computer Science,* vol. 276, no. 1, pp. 51-81. 2002.

[69] W. K. G. Assunção, T. E. Colanzi, S. R. Vergilio, and A. Pozo, "A multi-objective optimization approach for the integration and test order problem," *Information Sciences,* vol. 267, pp. 119-139. Elsevier, 2014.

[70] M. López-Ibánez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package, iterated race for automatic algorithm configuration," Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium2011.

[71] D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithm research: A history and analysis," Technical Report TR-98-03, Department of Electrical and Computer Engineering, Graduate School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio1998.

Part II

Papers

Paper A

STIPI: Using Search to Prioritize Test Cases Based on Multi-objectives Derived from Industrial Practice

Dipesh Pradhan, Shuai Wang, Shaukat Ali,
Tao Yue, Marius Liaaen

Published in the 28th International Conference on Testing Software and Systems (ICTSS), October 17-19, 2016.
DOI: 10.1007/978-3-319-47443-4_11

The layout has been revised.

Abstract

The importance of cost-effectively prioritizing test cases is undeniable in automated testing practice in industry. This paper focuses on prioritizing test cases developed to test product lines of Video Conferencing Systems (VCSs) at Cisco Systems, Norway. Each test case requires setting up configurations of a set of VCSs, invoking a set of test APIs with specific inputs, and checking statuses of the VCSs under test. Based on these characteristics and available information related with test case execution (e.g., number of faults detected), we identified that the test case prioritization problem in our particular context should focus on achieving high coverage of configurations, test APIs, statuses, and high fault detection capability as quickly as possible. To solve this problem, we propose a search-based test case prioritization approach (named STIPI) by defining a fitness function with four objectives and integrating it with a widely applied multi-objective optimization algorithm (named Non-dominated Sorting Genetic Algorithm II). We compared STIPI with random search (RS), Greedy algorithm, and three approaches adapted from literature, using three real sets of test cases from Cisco with four time budgets (25%, 50%, 75% and 100%). Results show that STIPI significantly outperformed the selected approaches and managed to achieve better performance than RS for on average 39.9%, 18.6%, 32.7% and 43.9% for the coverage of configurations, test APIs, statuses and fault detection capability, respectively.

Keywords: Test Case Prioritization; Search; Configurations; Test APIs.

1 Introduction

Testing is a critical activity for system or software development, through which system/software quality is ensured [1]. To improve the testing efficiency, a large number of researchers have been focusing on prioritizing test cases into an optimal execution order to achieve maximum effectiveness (e.g., fault detection capability) as quickly as possible [2-4]. In the industrial practice of automated testing, test case prioritization is even more critical because usually there is a limited budget (e.g., time) to execute test cases, and thus executing all available test cases at a given context is infeasible [1, 5].

Our industrial partner for this work is Cisco System, Norway, who develops product lines of Video Conferencing Systems (VCSs), which enable high quality conference meetings [4, 5]. To ensure the delivery of high quality VCSs to the market, test engineers of Cisco continually develop test cases to test software of VCSs under various hardware or software configurations, statuses (i.e., states) of VCSs with dedicated test APIs. A test case is typically composed of the following parts: 1) setting up test configurations of a set of VCSs under test; 2) invoking a set of test APIs of the VCSs; and 3) checking the statuses of the VCSs after invoking the test APIs to determine the success or failure of an execution of the test case. When executing test cases, several objectives need to be achieved, i.e.,

covering the maximum number of possible configurations, test APIs, statuses and detecting as many faults as possible. However, given a number of available test cases, it is often infeasible to execute all of them in practice due to a limited budget of execution time (e.g., ten hours), and it is therefore important to seek an approach for prioritizing the given test cases to cover maximum number of configurations, test APIs, statuses and detect faults as quickly as possible.

To address the above-mentioned challenge, we propose a search-based test case prioritization approach named *Search-based Test case prioritization based on Incremental unique coverage and Position Impact* (*STIPI*). *STIPI* defines a fitness function with four objectives to evaluate the quality of test case prioritization solutions, i.e., Configuration Coverage (*CC*), test API Coverage (*APIC*), Status Coverage (*SC*) and Fault Detection Capability (*FDC*), and integrates the fitness function with a widely-applied multi-objective search algorithm (i.e., Non-dominated Sorting Genetic Algorithm II) [6]. Moreover, we propose two prioritization strategies when defining the fitness function in *STIPI*: 1) *Incremental Unique Coverage*, i.e., for a specific test case, we only consider the incremental unique elements (e.g., test APIs) covered by the test case as compared with the elements covered by the already prioritized test cases; and 2) *Position Impact*, i.e., a test case with a higher execution position (i.e., scheduled to be executed earlier) has more impact on the quality of a prioritization solution. Notice that both of these strategies are defined to help search to achieve high criteria (i.e., *CC*, *APIC*, *SC* and *FDC*) as quickly as possible.

To evaluate *STIPI*, we chose five approaches for the comparison: 1) Random Search (*RS*) to assess the complexity of the problem; 2) *Greedy* approach; 3) One existing approach [7] and two modified approaches from the existing literature [8, 9]. The evaluation uses in total 211 test cases from Cisco, which are divided into three sets with varying complexity. Moreover, four different time budgets are used for our evaluation, i.e., 25%, 50%, 75% and 100% (100% refers to the total execution time of all the test cases in a given set). Notice that 12 comparisons were performed (i.e., three sets of test cases*four time budgets) for comparing *STIPI* with each approach, and thus in total 60 comparisons were conducted for the five approaches. Results show that *STIPI* significantly outperformed the selected approaches for 54 out of 60 comparisons (90%). In addition, *STIPI* managed to achieve higher performance than RS for on average 39.9% (configuration coverage), 18.6% (test API coverage), 32.7% (status coverage), and 43.9% (fault detection capability).

The remainder of the paper is organized as follows: Section 2 presents the context, a running example and motivation. *STIPI* is presented in Section 3 followed by experiment design (Section 4). Section 5 presents experiment results and overall discussion. Related work is discussed in Section 6, and we conclude the work in Section 7.

2 Context, Running Example and Motivation

Fig. A-1 presents a simplified context of testing VCSs (Systems Under Test (SUTs)), and Fig. A-2 illustrates (partial) configuration, test API and status information for testing a VCS. First, one VCS consists of one or more configuration variables (e.g., attribute *protocol* of class VCS in Fig. A-2), each of which can take two or more configuration variable values (e.g., literal *SIP* of enumeration *Protocol*). Second, a VCS holds one or more status variables defining the statuses of the VCS (e.g., *NumberofActiveCalls*), and each status variable can have two or more status variable values (e.g., *NumberofActiveCalls* taking values of 0, 1, 2, 3, and 4). Third, testing a VCS requires employing one or more test API commands (e.g., *dial*), each of which includes zero or more test API parameters (e.g., *callType* for *dial*). Each test API parameter can take two or more test API parameter values (e.g., *Video* and *Audio* for *CallType*).

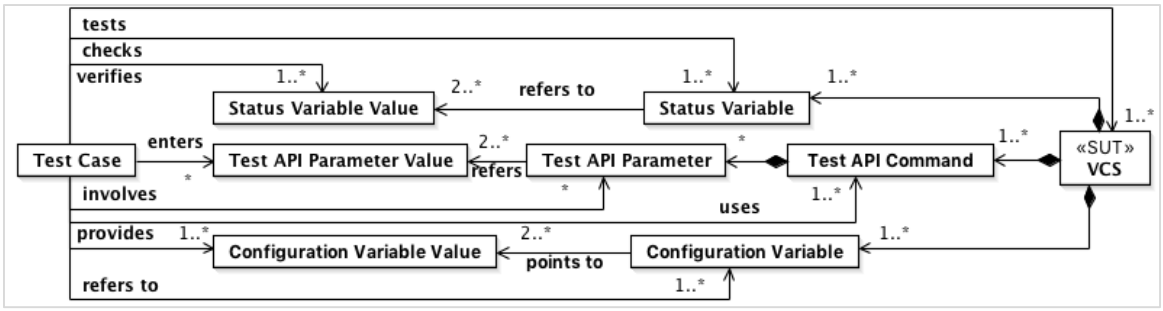

Fig. A-1. A simplified context of testing VCSs

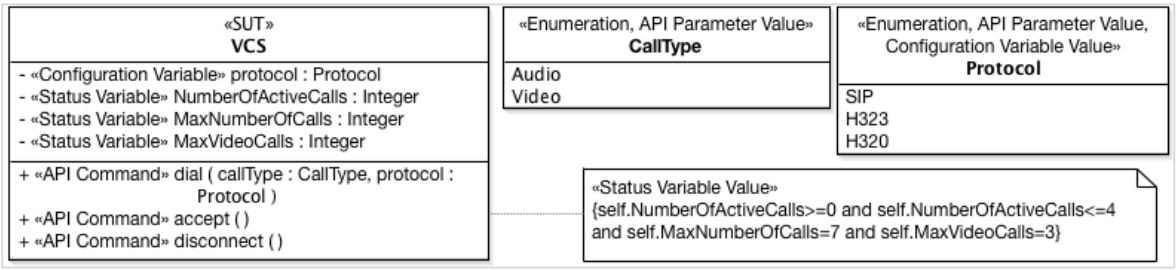

Fig. A-2. Partial configuration, status and test API Information for testing a VCS

Fig. A-3 illustrates the key steps of a test case for testing VCSs. First, a test case configures one or more VCSs by assigning values to configuration variables. For example, the test case shown in Fig. A-3 configures the configuration variable *protocol* with *SIP* (line 1). Second, a test API command is invoked with appropriate values assigned to its input parameters, if any. For example, the test case in Fig. A-3 invokes the test API command *dial* consisting of the two test API parameter values: *Video* for *callType* and *SIP* for *protocol*). Third, the test case checks the actual statuses of VCSs. For example, the test case in Fig. A-3 checks the status of the VCS to see if *NumberOfActiveCalls* equals to 1.

```
1. protocol = SIP   //Configure the configuration variable
2. dial(Video, SIP) //Employ test API command dial and assigning
                      values to parameters: callType and protocol
3. accept           //Employ test API command with no parameters
4. assert (NumberOfActiveCalls=1,MaxNumberOfCalls=1,
          MaxVideoCalls =1) //Check values of the status variables
5. disconnect       //Employ test API command with no parameters
6. assert(NumberofActiveCalls=0) //Check status
```

Fig. A-3. An excerpt of a sanitized and simplified test case

In the context of testing VCSs, test case prioritization is a critical task since it is practically infeasible to execute all the available test cases within a given time budget (e.g., five hours). Therefore, it is essential to cover maximum configurations (i.e., configuration variables and their values), test APIs (i.e., test API commands, parameters and their values) and statuses (i.e., status variables and their values), and detect faults as quickly as possible. For instance, Table A-1 lists five test cases (T_1,…, T_5) with the information about configurations, test APIs and statuses. The test case in Fig. A-3 is represented as T_1 in Table A-1, which 1) sets the configuration variable *protocol* as *SIP*; 2) uses three test API commands: *dial* with two parameters (*callType*, *protocol*), *accept* and *disconnect*; and 3) checks values of three status variables (e.g., *MaxVideoCalls*).

Table A-1. Illustrating Test Case Prioritization*

Test Case	Configuration	Test API				Status		
	protocol	dial		accept	disco nnect	SV_1	SV_2	SV_3
		callType	protocol					
T_1	SIP	Video	SIP	✓	✓	0, 1	1	1
T_2	SIP	Audio	SIP	✓	✓	0, 1	1	0
T_3	SIP	Audio	SIP	✓		1	1	0
T_4	H323	Audio	H323	✓		0, 1, 2	2	0
T_5	H320	Audio	H320	✓		1	1	1

* *SV_1: NumberOfActiveCalls, SV_2: MaxNumberOfCalls, SV_3: MaxVideoCalls.*

Notice that the five test cases in Table A-1 can be executed in 325 orders (i.e., $C(5,1)\times1! + C(5,2)\times2! + , \ldots, + C(5,5)\times5!$). When there is a time budget, each particular order can be considered as a prioritization solution. Given two prioritization solutions $s_1 = \{T_5, T_1, T_4, T_2, T_3\}$, $s_2 = \{T_1, T_3, T_5, T_2, T_4\}$, one can observe that s_1 is better than s_2 since the first three test cases in s_1 can cover all the configuration variables and their values, test API commands, test API parameters, test API parameter values, status variables and status variable values, while s_2 needs to execute all the five test cases to achieve the same coverage as s_1. Therefore, it is important to seek an efficient approach to find an optimal order for executing a given number of test cases to achieve high coverage of configurations, test APIs and statuses, and detect faults as quickly as possible, which forms the motivation of this work.

3 *STIPI*: Search-based Test case prioritization based on Incremental unique coverage and Position Impact

This section presents the problem representation (Section 3.1), four defined objectives, fitness function (Section 3.2) and solution encoding (Section 3.3).

3.1 Basic Notations and Problem Representation

Basic Notations. We provide the basic notations as below used throughout the paper.

$T = \{T_1, T_2, \dots, T_n\}$ represents a set of *n* test cases to be prioritized.

$ET = \{et_1, et_2, \dots, et_n\}$ refers to the execution time for each test case in T.

$CV = \{cv_1, cv_2, \dots, cv_{mcv}\}$ represents the configuration variables covered by T. For each cv_i, CVV_i refers to the configuration variable values: $CVV_i = \{cvv_{i1}, \dots, cvv_{icvv}\}$. $mcvv$ is the total number of unique values for all the configuration variables, which can be calculated as: $mcvv = |(\bigcup_{i=1}^{mcv} CVV_i)|$.

$AC = \{ac_1, ac_2, \dots, ac_{mac}\}$ represents a set of test API commands covered by T. For each ac_i, AP_i denotes the test API parameters: $AP_i = \{ap_{i1}, \dots, ap_{iap}\}$. map is the total number of unique test API parameters, calculated as: $map = |(\bigcup_{i=1}^{mac} AP_i)|$. For each ap_i, AV_i refers to the test API parameter values: $AV_i = \{av_{i1}, \dots, av_{iav}\}$. mav is the total number of unique test API parameter values, i.e., $mav = |(\bigcup_{i=1}^{map} AV_i)|$.

$SV = \{sv_1, sv_2, \dots, sv_{msv}\}$ represents a set of status variables covered by T. For each sv_i, SVV_i refers to the status variable values: $SVV_i = \{svv_{i1}, \dots, svv_{isvv}\}$. $msvv$ is the total number of unique status variable values, calculated as: $msvv = |(\bigcup_{i=1}^{msv} SVV_i)|$.

$Effect = \{effect_1, \dots, effect_{neffect}\}$ defines a set of effectiveness measures.

$S = \{s_1, s_2, \dots, s_{ns}\}$ represents a set of potential solutions, such that $ns = C(n,1)\times 1! + C(n,2)\times 2! + , \dots, + C(n,n)\times n!$. Each solution s_j consists of a set of prioritized test cases in T: $s_j = \{T_{j1}, \dots, T_{jn}\}$, where $T_{ji} \in T$ refers to the test case with the execution position i in the prioritized solution s_j. Note that it is possible for the maximum number of test cases in s_j (i.e., jn) to be less than the total number of test cases in T, since only a subset of T is prioritized during limited budget (e.g., time).

Problem Representation. We aim to prioritize the test cases in T in two contexts: 1) 100% time budget and 2) less than 100% time budget (i.e., time-aware [1]). Therefore, we formulate the test case prioritization problem as follows: a) search a solution s_k with nk test cases from the total number of ns solutions in S to obtain the highest effectiveness; and b) a test case T_{jr} in a particular solution (e.g., s_j) with a higher position p has more influence for $Effect$ than the test case with a lower position q.

1) With 100% time budget:

$\forall_{i=1\ to\ neffect} \forall_{j=1\ to\ ns}\ Effect\ (s_k, effect_i) \geq Effect\ (s_j, effect_i)$

$\vee\ effect_i(T_{jr}, p) > \forall_{q \geq (p+1)} effect_i(T_{jr}, q)$.

where, $effect_i(T_{jr}, p)$ and $effect_i(T_{jr}, q)$ refer to the effectiveness measure i for a test case T_{jr} at position p and q, respectively for a particular solution s_j. $Effect\ (s_k, effect_i)$ and $Effect\ (s_j, effect_i)$ returns the effectiveness measure i for solutions s_k, s_j respectively.

2) With a time budget tb less than 100% time budget:
$\forall_{i=1\ to\ neffect} \forall_{j=1\ to\ ns}\ Effect\ (s_k, effect_i) \geq Effect\ (s_j, effect_i)$
$\vee \sum_{l=1}^{nk} ET_l \leq tb, effect_i(T_{jr}, p) > \forall_{q \geq (p+1)} effect_i(T_{jr}, q)$.

3.2 Fitness Function

Recall that we aim at maximizing the overall coverage for configuration, test API and status, and detect faults as quickly as possible (Section 2). Therefore, we define four objective functions for the fitness function to guide the search towards finding optimal solutions, which are presented in details as below.

Maximize Configuration Coverage (*CC*). *CC* measures the overall configuration coverage of a solution s_j with jn number of test cases, which is composed of Configuration Variable Coverage (*CVC*) and Configuration Variable Values Coverage (*CVVC*). We can calculate *CVC* and *CVVC* for s_j as:

$CVC_{s_j} = \frac{\sum_{i=1}^{jn} UCV_{T_{ji}} \times \frac{n-i+1}{n}}{mcv}$, and $CVVC_{s_j} = \frac{\sum_{i=1}^{jn} UCVV_{T_{ji}} \times \frac{n-i+1}{n}}{mcvv}$, where *mcv* and *mcvv* represent the total number of unique Configuration Variables (*CV*) and Configuration Variable Values (*CVV*) respectively covered by the total test cases in T (e.g., in Table A-1, $mcvv = 3$). Moreover, we propose two prioritization strategies for calculating CVC and $CVVC$. The first one is *Incremental Unique Coverage*, i.e., $UCV_{T_{ji}}$ and $UCVV_{T_{ji}}$ representing the number of incremental unique *CV* and *CVV* covered by T_{ji} (Section 3.1). For example, in Table A-1, for one test case prioritization solution $s_1 = \{T_5, T_1, T_4, T_2, T_3\}, UCVV_{T_5}$ is 1 since T_5 is in the first execution position and covers one *CVV* (i.e., *H320*). $UCVV_{T_1}$ and $UCVV_{T_4}$ are at the second and third position, and cover one *CVV* each (i.e., *SIP*, *H323*). However, $UCVV_{T_2}$ and $UCVV_{T_3}$ are 0, since they are already covered by $UCVV_{T_1}$. This strategy is defined since test case prioritization in our case concerns how many configurations, test APIs, and statuses can be covered rather than how many times they can be covered.

The second prioritization strategy is *Position Impact*, which is calculated as $\frac{n-i+1}{n}$, where n is the total number of test cases, and i is a specific execution position in a prioritization solution. Thus, test cases with higher execution positions have higher impact on the quality of a prioritization solution, which fits the scope of test case prioritization that aims at achieving higher criteria as quickly as possible. For instance, using this strategy,

$CVVC$ for s_1 is: $CVVC_{s_1} = \frac{1\times\frac{5}{5}+1\times\frac{4}{5}+1\times\frac{3}{5}+0\times\frac{2}{5}+0\times\frac{1}{5}}{3} = 0.8$. Moreover, CC for s_j is represented as:$CC_{s_j} = \frac{CVC_{s_j}+CVVC_{s_j}}{2}$. A higher value of CC shows a higher coverage of configuration.

Maximize Test API Coverage (*APIC*). *APIC* measures the overall test API coverage of a solution s_j with jn number of test cases. It consists of three sub measures: Test API Command Coverage (*ACC*), Test API Parameter Coverage (*APC*), and Test API parameter Value Coverage (*AVC*). *ACC*, *APC* and *AVC* can be calculated as below:

$$ACC_{s_j} = \frac{\sum_{i=1}^{jn} UAC_{T_{ji}} \times \frac{n-i+1}{n}}{mac}, APC_{s_j} = \frac{\sum_{i=1}^{jn} UAP_{T_{ji}} \times \frac{n-i+1}{n}}{map}, AVC_{s_j} = \frac{\sum_{i=1}^{jn} UAV_{T_{ji}} \times \frac{n-i+1}{n}}{mav}.$$

Similarly, the same two strategies (i.e., *Incremental Unique Coverage* and *Position Impact*) are applied for calculating ACC, APC and AVC, where $UAC_{T_{ji}}$, $UAP_{T_{ji}}$ and $UAV_{T_{ji}}$ denotes the number of unique test API commands (*AC*), test API parameters (*AP*), and test API parameter values (*AV*) respectively covered by T_{ji} (Section 3.1). They are measured similar as for $UCVV_T$ in $CVVC$. mac, map, and mav refer to the total number of unique *AC*, *AP*, and *AV* covered by the total number of test cases as explained for $mcvv$ in *CVVC*. The *APIC* for s_j is represented as: $APIC_{s_j} = \frac{ACC_{s_j}+APC_{s_j}+AVC_{s_j}}{3}$. A higher value of *APIC* shows a higher coverage of test APIs.

Maximize Status Coverage (*SC*). *SC* measures the total status coverage of a solution s_j. It consists of two sub measures: Status Variable Coverage (*SVC*) and Status Variable Value Coverage (*SVVC*), calculated as follow: $SVC_{s_j} = \frac{\sum_{i=1}^{jn} USV_{T_{ji}} \times \frac{n-i+1}{n}}{msv}$, $SVVC_{s_j} = \frac{\sum_{i=1}^{jn} USVV_{T_{ji}} \times \frac{n-i+1}{n}}{msvv}$. Similarly, $USV_{T_{ji}}$ and $USVV_{T_{ji}}$ are the number of unique Status Variables (*SV*) and Status Variable Values (*SVV*) respectively covered by T_{ji} (Section 3.1), which are measured similar as $UCVV_T$ in $CVVC$. msv and $msvv$ represent the total number of unique *SV* and *SVV* respectively measured similar as for $mcvv$ in *CVVC*. The *SC* for s_j is represented as: $SC_{s_j} = \frac{SVC_{s_j}+SVVC_{s_j}}{2}$, with a higher value indicating a higher status coverage, and therefore representing a better solution.

Maximize Fault Detection Capability (*FDC*). In the context of Cisco, *FDC* is defined as the detected number of faults for test cases in a solution s_j [4, 5, 10-12]. The *FDC* for a test case T_{ji} is calculated as: $FDC_{T_{ji}} = \frac{Number\ of\ times\ that\ T_{ji}\ found\ a\ fault}{Number\ of\ times\ that\ T_{ji}\ was\ executed}$. Notice that the *FDC* of T_{ji} is calculated based on the historical information of executing T_{ji}. For example, if tc_i was executed 10 times, and it detected fault 4 times, the *FDC* for tc_i is 0.4. We calculate *FDC* for a solution s_j as: $FDC_{s_j} = \frac{\sum_{i=1}^{jn} FDC_{T_{ji}} \times \frac{n-i+1}{n}}{mfdc}$. $FDC_{T_{ji}}$ denotes the *FDC* for

a T_{ji} , $mfdc$ represents the sum of all FDC of test cases, and a higher value of FDC implies a better solution. Notice that we cannot apply the *incremental unique coverage* strategy for calculating FDC_{s_j} since the relations between faults and test cases are not known in our case (i.e., we only know whether the test cases can detect faults after executing it for a certain number of times rather than having access to the detailed faults detected).

3.3 Solution Representation

The test cases in T are encoded as an array $A = \{v_1, v_2 \dots v_n\}$, where each variable v_i represents one test case in T, and holds a unique value from 0 to 1. We prioritize the test cases in TS by sorting the variables in A in a descending order from higher to lower, such that 1 is the highest, and 0 is the lowest order. Initially, each variable in A is assigned a random value between 0 and 1, and during search our approach returns solutions with optimal values for A guided by the fitness function defined in Section 3.2. In terms of time-aware test case prioritization (i.e., with a time budget less than 100%), we pick the maximum number of test cases that fit the given time budget. For example, in Table A-1 for $TS = \{T_1, \dots, T_5\}$ with A as $\{0.6, 0.2, 0.4, 0.9, 0.3\}$ and the execution time (recorded as minutes) as $ET = \{4, 5, 6, 4, 3\}$, the prioritized test cases are $\{T_4, T_1, T_3, T_5, T_2\}$ based on our encoding way for test case prioritization. If we have a time budget of 11 minutes, the first two test cases (in total 8 minutes for execution) are first added to the prioritized solution s_j, and there are 3 minutes left, which is not sufficient for executing T_3 (6 minutes). Thus, T_3 is not added into s_j, and the next test case is evaluated to see if the total execution time can fit the given time budget. T_5 with three minutes will be added into s_j, since the inclusion of T_5 will not make the total execution time exceed the time budget. Therefore, the new prioritized solution will be $\{T_4, T_1, T_5\}$.

Moreover, we integrate our fitness function with a widely applied multi-objective search algorithm named Non-dominated Sorting Genetic Algorithm (NSGA-II) [6, 13, 14]. The tournament selection operator [6] is applied to select individual solutions with the best fitness for inclusion into the next generation. The crossover operator is used to produce offspring solutions from the parent solutions by swapping some of the parts (e.g., test cases in our context) of the parent solutions. The mutation operator is applied to randomly change the values of one or more variables (e.g., in our context, each variable represents a test case) based on the pre-defined mutation probability, e.g., $\frac{1}{(total\ number\ of\ test\ cases)}$ in our context.

4 Empirical Study Design

4.1 Research Questions

RQ1: Is *STIPI* effective for test case prioritization as compared with *RS* (i.e., random prioritization)? We compare *STIPI* with *RS* for four time budgets: 100% (i.e., total execution time of all the test cases in a given set), 75%, 50% and 25%, to assess the complexity of the problem such that the use of search algorithms is justified.

RQ2: Is *STIPI* effective for test case prioritization as compared with four selected approaches, in the contexts of four time budgets: 100%, 75%, 50% and 25%?

RQ2.1: Is *STIPI* effective as compared with the Greedy approach (a local search approach)? ***RQ2.2***: Is *STIPI* effective as compared with the approach used in [7] (named as *A1* in this paper)? Notice that we chose *A1* since it also proposed a strategy to give higher importance to test cases with higher execution positions.

RQ2.3: Is *STIPI* effective as compared with the modified version of the approach proposed in [8] (named as *A2* in this paper)? We chose *A2* since it combines the Average Percentage of Faults Detected (*APFD*) metric and NSGA-II for test case prioritization without considering time budget. We modified it by defining Average Percentage of Configuration Coverage (*APCC*), Average Percentage of test API Coverage (*APAC*) and Average Percentage of Status Coverage (*APSC*) (Section 4.3) for assessing the quality of prioritization solutions for configurations, test APIs and statuses.

RQ2.4: Is *STIPI* effective as compared with the modified version of the approach in [9] (named as *A3* in this paper)? We chose *A3* since 1) it combines the *ADFD* with cost ($APFD_c$) metric and NSGA-II for addressing time-aware test case prioritization problem. We revised *A3* by defining Average Percentage of Configuration Coverage with cost ($APCC_c$), Average Percentage of test API Coverage with cost ($APAC_c$) and Average Percentage of Status Coverage with cost ($APSC_c$). For illustration, we provide a formula for Average Percentage of Configuration Variable Value Coverage with cost ($APCVVC_c$) that is a sub-metric for $APCC_c$ as: $APCVVC_c = \frac{\sum_{i=1}^{mcvv}(\sum_{k=TCVV_i}^{jn} et_k - \frac{1}{2}et_{TCVV_i})}{\sum_{k=1}^{jn} et_k \times mcvv}$. For a solution s_j with jn test cases, $TCVV_i$ is the first test case from s_j that covers CVV_i (i.e., the i^{th} configuration variable value), $mcvv$ is the total number of unique configuration variable value, and et_k is the execution time for k^{th} test case. Notice that the detailed formulas for $APCC_c$, $APAC_c$ and $APSC_c$ can be consulted in our technical report in [15].

We also compare the running time of *STIPI* with all the five chosen approaches, since *STIPI* is invoked very frequently (e.g., more than 50 times per day) in our context, i.e., the test cases require to be prioritized and executed often. Therefore, it would be practically infeasible if it takes too much time to apply *STIPI*.

4.2 Experiment Tasks

As shown in Table A-2 (*Experiment Task* column), we designed two tasks (T_1, T_2) for addressing *RQ1-RQ2*. The task T_1 is designed to compare *STIPI* with *RS* for the four time budgets (i.e., 100%, 75%, 50% and 25%) and three sets of test cases (i.e., 100, 150 and 211). Similarly, the task T_2 is designed to compare *STIPI* with the other four test case prioritization approaches, which is divided into four sub-tasks for comparing *Greedy*, *A1*, *A2* and *A3*, respectively.

Moreover, we employed 211 real test cases from Cisco for evaluation by dividing it into three sets with varying complexity (*#Test Cases* column in Table A-2). For the first set, we used all the 211 test cases. For the second set, we used 100 random test cases from the 211 test cases. Finally, for the third set, we used the 150 test cases by choosing 111 test cases not selected in the second set (i.e., 100) and 39 random test cases from the second set. Notice that the goal for using three test case sets is to evaluate our approach with test datasets with different complexity.

Table A-2. Overview of the Experiment Design

RQ	**Experiment Task**		**#Test Cases**	**Time Budget%**	**Evaluation Metric (*EM*)**	**Quality Indicator**	**Statistical Test**
1	T_1: *STIPI* vs. *RS*		100 150 211	100	*APCC, APAC, APSC*	-	Vargha and Delaney $\hat{A}_{12}$Mann-Whitney U Test
				25, 50, 75	$APCC_p$, $APAC_p$, $APSC_p$, *MFDC*	-	
2	$T_{2.1}$	*STIPI* vs. *Greedy*		100	*APCC, APAC, APSC*	-	
				25, 50, 75	$APCC_p$, $APAC_p$, $APSC_p$, *MFDC*	-	
	$T_{2.2}$	*STIPI* vs. *A1*		100	*APCC, APAC, APSC*	*Hypervolume(HV)*	
				25, 50, 75	$APCC_p$, $APAC_p$, $APSC_p$, *MFDC*		
	$T_{2.3}$	*STIPI* vs. *A2*		100	*APCC, APAC, APSC*		
				25, 50, 75	$APCC_p$, $APAC_p$, $APSC_p$		
	$T_{2.4}$	*STIPI* vs. *A3*		100	*APCC, APAC, APSC*		
				25, 50, 75	$APCC_p$, $APAC_p$, $APSC_p$		

4.3 Evaluation Metrics

To answer the *RQ*s, we defined in total seven *EMs* (Table A-3). Six are used to assess how fast the configurations, test APIs and statuses can be covered: 1) Average Percentage Configuration Coverage (*APCC*), 2) Average Percentage test API Coverage (*APAC*), 3) Average Percentage Status Coverage (*APSC*), 4) Average Percentage Configuration Coverage that penalizes missing configuration ($APCC_p$), 5) Average Percentage test API Coverage that penalizes missing test API ($APAC_p$) and 6) Average Percentage Status Coverage with penalization for missing status ($APSC_p$). We defined *APCC*, *APAC* and *APSC* for test case prioritization with 100% time budget based on the *APFD* metric [8, 16]. For example, for a solution s_j with *jn* test cases and total number of test cases *n* from *T* (a given number of test cases), TCV_1 is the first test case from s_j that covers CV_1 for the sub metric *APCVC* in Table A-3 (Section 3.1). Notice that *n* and *jn* are equal when there is 100% time budget.

Table A-3. Different metrics for evaluating the approaches*

EC	Time Budget%	*EM*	Sub Metric		Formula
			Name	Formula	
Con	100	$APCC$	$APCVC$	$1-\frac{TCV_1+TCV_2+\ ...\ +TCV_{mcv}}{n\times mcv}+\frac{1}{2n}$	$APCC=\frac{APCVC+APCVVC}{2}$
			$APCVVC$	$1-\frac{TCVV_1+TCVV_2+\ ...\ +TCVV_{mcvv}}{n\times mcvv}+\frac{1}{2n}$	
	25 50 75	$APCC_p$	$APCVC_p$	$1-\frac{\sum_{cv=1}^{mcv} reveal(cv,s_j)}{jn\times mcv}+\frac{1}{2jn}$	$APCC_p=\frac{APCVC_p+APCVVC_p}{2}$
			$APCVVC_p$	$1-\frac{\sum_{cvv=1}^{mcvv} reveal(cvv,s_j)}{jn\times mcvv}+\frac{1}{2jn}$	
API	100	$APAC$	$APACC$	$1-\frac{TAC_1+TAC_2+\ ...\ +TCA_{mac}}{n\times mac}+\frac{1}{2n}$	$APAC=\frac{APACC+APAPC+APAVC}{3}$
			$APAPC$	$1-\frac{TAP_1+TAP_2+\ ...\ +TAP_{map}}{n\times map}+\frac{1}{2n}$	
			$APAVC$	$1-\frac{TAV_1+TAV_2+\ ...\ +TAV_{mav}}{n\times mav}+\frac{1}{2n}$	
	25 50 75	$APAC_p$	$APACC_p$	$1-\frac{\sum_{ac=1}^{mac} reveal(ac,s_j)}{jn\times mac}+\frac{1}{2jn}$	$APAC_p=\frac{APACC_p+APAPC_p+APAVC_p}{3}$
			$APAPC_p$	$1-\frac{\sum_{ap=1}^{map} reveal(ap,s_j)}{jn\times map}+\frac{1}{2jn}$	
			$APAVC_p$	$1-\frac{\sum_{av=1}^{mav} reveal(av,s_j)}{jn\times mav}+\frac{1}{2jn}$	
Stat	100	$APSC$	$APSVC$	$1-\frac{TSV_1+TSV_2+\ ...\ +TSV_{msv}}{n\times msv}+\frac{1}{2n}$	$APSC=\frac{APSVC+APSVVC}{2}$
			$APSVVC$	$1-\frac{TSVV_1+TSVV_2+\ ...\ +TSVV_{msvv}}{n\times msvv}+\frac{1}{2n}$	
	25 50 75	$APSC_p$	$APSVC_p$	$1-\frac{\sum_{sv=1}^{msv} reveal(sv,s_j)}{jn\times msv}+\frac{1}{2jn}$	$APSC_p=\frac{APSVC_p+APSVVC_p}{2}$
			$APSVVC_p$	$1-\frac{\sum_{svv=1}^{msvv} reveal(svv,s_j)}{jn\times msvv}+\frac{1}{2jn}$	
FDC	25,50,75	$MFDC$	-	-	$MFDC=\frac{\sum_{i=1}^{jn} FDC_{T_i}}{\sum_{k=1}^{n} FDC_{T_k}}\times 100\%$

**EC*: Evaluation Criteria, Con: Configuration, API: Test API, Stat: Status.

4.4 Quality Indicator, Statistical Tests and Parameter Settings

When comparing the overall performance of multi-objective search algorithms (e.g., NSGA-II [6]), it is common to apply quality indicators such as hypervolume (*HV*). Following the guideline in [10], we employ *HV* based on the defined *EM*s to address *RQ2.2-RQ2.4* (i.e., tasks $T_{2.2}-T_{2.4}$ in Table A-2). *HV* calculates the volume in the objective space covered by members of a non-dominated set of solutions (i.e., Pareto front) produced by search algorithms for measuring both convergence and diversity [17]. A higher value of *HV* indicates a better performance of the algorithm.

The Vargha and Delaney $\hat{A}_{12}$ statistics [18] and Mann-Whitney U test are used to compare the EMs (T_1 and T_2), and *HV* ($T_{2.2}-T_{2.4}$), as shown in Table A-2 by following the guidelines in [19]. The Vargha and Delaney $\hat{A}_{12}$ statistics is a non-parametric effect size measure, and Mann-Whitney U test tells if results are statistically significant [20]. For two

algorithms A and B, A has better performance than B if $\hat{A}_{12}$ is greater than 0.5, and the difference is significant if p-value is less than 0.05.

Notice that *STIPI*, *A1*, *A2* and *A3* are all combined with NSGA-II. Since tuning parameters to different settings might result in different performance of search algorithms, standard settings are recommended [19]. We used standard settings (i.e., population size=100, crossover rate=0.9, mutation rate=1/(number of test cases)) as implemented in jMetal [21]. The search process is terminated when the fitness function has been evaluated for 50,000 times. Since *A2* does not support prioritization with a time budget, we collect the maximum number of test cases that can fit a given time budget.

5 Results, Analysis and Discussion

5.1 *RQ1*: Sanity Check (*STIPI* vs. RS)

Results in Table A-4 and Table A-5 show that on average *STIPI* is higher than *RS* for all the *EM*s across the three sets of test cases. Moreover, for the three test sets using four time budgets, *STIPI* managed to achieve higher performance than RS for on average 39.9% (configuration coverage), 18.6% (test API coverage), 32.7% (status coverage), and 43.9% (*FDC*). In addition, results of the Vargha and Delaney statistics and the Mann Whitney U test show that *STIPI* significantly outperformed *RS* for all the *EM*s since all the values of $\hat{A}_{12}$ are greater than 0.5 and all the p-values are less than 0.05.

Table A-4. Average Values of the *EM*s with 100% and 75% Time Budget*

#*T*	100% time budget							75% time budget						
	EM	***RS***	**Gr**	***A1***	***A2***	***A3***	***STIPI***	***EM***	***RS***	**Gr**	***A1***	***A2***	***A3***	***STIPI***
100		0.7	0.76	0.75	0.77	0.75	0.77		0.63	0.71	0.73	0.74	0.73	0.74
150	***CC***	0.68	0.84	0.8	0.79	0.75	0.79	CC_p	0.60	0.81	0.69	0.72	0.73	0.77
211		0.74	0.83	0.83	0.85	0.81	0.85		0.67	0.76	0.79	0.80	0.79	0.81
100		0.83	0.74	0.85	0.85	0.84	0.86		0.78	0.70	0.83	0.82	0.84	0.83
150	***AC***	0.78	0.64	0.83	0.86	0.85	0.86	AC_p	0.72	0.57	0.75	0.81	0.83	0.84
211		0.82	0.67	0.85	0.89	0.89	0.89		0.77	0.56	0.83	0.87	0.87	0.88
100		0.73	0.65	0.76	0.82	0.76	0.82		0.67	0.60	0.73	0.79	0.79	0.81
150	***SC***	0.74	0.62	0.8	0.85	0.83	0.85	SC_p	0.68	0.56	0.71	0.80	0.81	0.83
211		0.78	0.64	0.79	0.85	0.82	0.85		0.72	0.56	0.79	0.84	0.85	0.86
100	-	-	-	-	-	-	-		0.78	0.79	0.91	-	-	0.89
150	-	-	-	-	-	-	-	***MF***	0.79	0.80	0.70	-	-	0.87
211	-	-	-	-	-	-	-		0.77	0.63	0.91	-	-	0.90

*T: Test Case, Gr: Greedy, *CC*: *APCC*, *AC*: *APAC*, *SC*: *APSC*, CC_p: $APCC_p$, AC_p: *APACp*, SC_p: *APSCp*, *MF*: *MFDC*.

5.2 *RQ2*: Comparison with the selected approaches

We compared *STIPI* with *Greedy*, *A1*, *A2* and *A3* using the statistical tests (Vargha and Delaney statistics and Mann Whitney U test) for the four time budgets (25%, 50%, 75% and 100%), and the three sets of test cases (i.e., 100, 150, 211). Results are summarized in

Fig. A-4. For example, the first bar (i.e., *Gr*) in Fig. A-4 refers to the comparison between *STIPI* and *Greedy* for the 100% time budget where *A= STIPI* and *B=Greedy*. *A>B* means the percentage of *EM*s for which *STIPI* has significantly better performance than *Greedy* ($\hat{A}_{12} > 0.5$ && $p < 0.05$), *A<B* means the opposite ($\hat{A}_{12} < 0.5$ && $p < 0.05$), and *A=B* implies there is no significant difference in performance ($p \geq 0.05$).

Table A-5. Average Values of the *EM*s with 25% and 50% Time Budget

EM	***#T***	**25% time budget**						**50% time budget**					
		RS	**Gr**	***A1***	***A2***	***A3***	***STIPI***	***RS***	**Gr**	***A1***	***A2***	***A3***	***STIPI***
$APCC_p$	100	0.37	0.30	0.55	0.51	0.62	0.66	0.52	0.65	0.65	0.67	0.70	0.73
	150	0.35	0.59	0.52	0.45	0.66	0.71	0.50	0.81	0.74	0.63	0.72	0.74
	211	0.42	0.43	0.63	0.56	0.69	0.71	0.52	0.53	0.65	0.67	0.70	0.73
$APAC_p$	100	0.56	0.26	0.70	0.61	0.74	0.70	0.71	0.61	0.79	0.77	0.81	0.81
	150	0.50	0.35	0.59	0.55	0.74	0.75	0.64	0.54	0.76	0.74	0.81	0.82
	211	0.58	0.33	0.71	0.65	0.77	0.75	0.71	0.52	0.79	0.81	0.85	0.85
$APSC_p$	100	0.42	0.14	0.59	0.55	0.70	0.66	0.57	0.51	0.68	0.72	0.76	0.76
	150	0.44	0.33	0.54	0.53	0.73	0.74	0.52	0.53	0.65	0.67	0.70	0.73
	211	0.48	0.24	0.66	0.62	0.78	0.77	0.63	0.52	0.74	0.78	0.84	0.85
MFDC	100	0.30	0.06	0.55	-	-	0.50	0.54	0.45	0.77	-	-	0.78
	150	0.30	0.19	0.40	-	-	0.63	0.55	0.74	0.75	-	-	0.76
	211	0.29	0.09	0.52	-	-	0.44	0.53	0.48	0.75	-	-	0.76

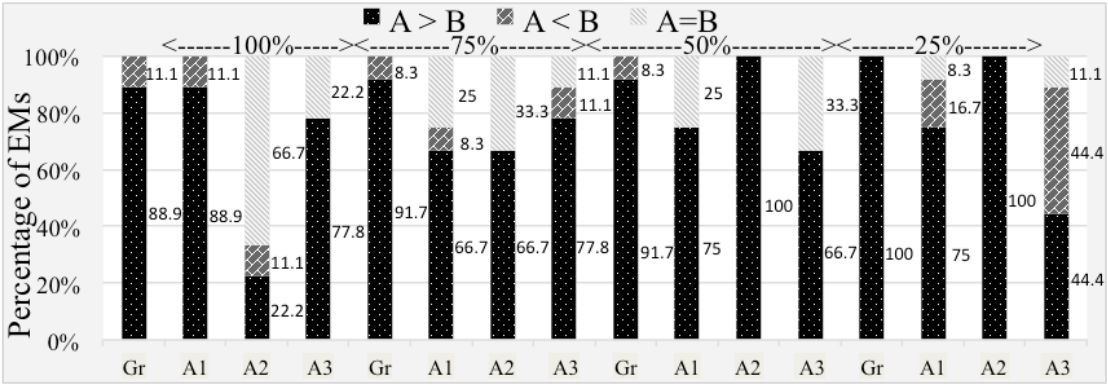

Fig. A-4. Results of comparing *STIPI* with *Greedy*, *A1*, *A2* and *A3* for *EM*s.

***RQ2.1* (*STIPI* vs. *Greedy*).** From Table A-4 and Table A-5, we can observe that the average values of *STIPI* are higher than *Greedy* for 93.3% (42/45)[1] *EM*s across the three sets of test cases with the four time budgets. Moreover, from Fig. A-4, we can observe *STIPI* performed significantly better than *Greedy* for an average of 93.1% for the four time budgets (i.e., 88.9% for 100%, 91.7% for 75%, 91.7% for 50%, and 100% for 25% time budget). Detailed results are available in [15].

***RQ2.2* (*STIPI* vs. *A1*).** Based on Table A-4 and Table A-5, we can see that *STIPI* has a higher average value than *A1* for 82.2% (37/45) *EM*s, and *STIPI* performed significantly

[1] An *EM* has one average value for one set of test case with one time budget (Table A-4 and Table A-5). Thus, for 100% time budget with 3 *EMs* there are 9 values, and 45 average values for 4 time budgets and 4 EMs for other 3 time budgets.

better than *A1* for an average of 76.4% *EM*s across the four time budgets, while there was no difference in performance for 14.6% from Fig. A-4. Fig. A-5 shows that for *HV*, *STIPI* outperformed *A1* for all the three sets of test cases with the four time budgets, and such better results are statistically significant. Detailed results are in [15].

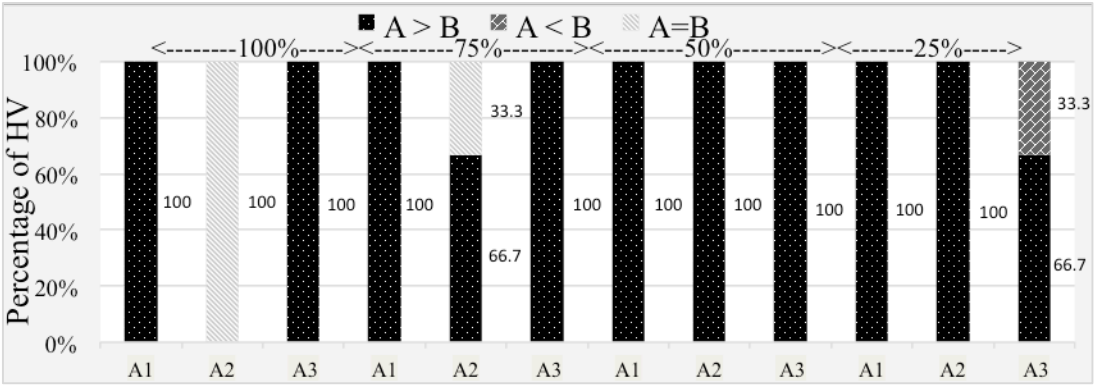

Fig. A-5. Results of comparing *STIPI* with *A1*, *A2* and *A3* for *HV*

***RQ2.3* (*STIPI* vs. *A2*).** *RQ2.3* is designed to compare *STIPI* with the approach *A2* (Section 4.1). Table A-4 shows that the two approaches had similar average for *EMs* with 100% time budget. Moreover, for 100% time budget, there was no significant difference in the performance between *STIPI* and *A2* in terms of *EMs* and *HV* (Fig. A-4 and Fig. A-5). However, when considering the time budgets of 25%, 50% and 75%, *STIPI* had a higher performance for 96.3% (26/27) *EM*s (Table A-4 and Table A-5). Furthermore, the statistical tests in Fig. A-4 and Fig. A-5 show that *STIPI* significantly outperformed *A2* for an average of 88.9% *EM*s and *HV* values across the three time budgets (25%, 50%, 75%), while there was no significant difference for 11.1%.

***RQ2.4* (*STIPI* vs. *A3*).** Based on the results (Table A-4 and Table A-5), *STIPI* held a higher average values for 75% (27/36) *EM* values for the four time budgets and three sets of test cases. For 100%, 75%, and 50%, we can observe from Fig. A-4 that *STIPI* performed significantly better than *A3* for an average of 74.1% *EM*s, while there was no significant difference for 22.2%. For the 25% time budget, there was no statistically significant difference in terms of *EM*s for *STIPI* and *A3*. However, when comparing the *HV* values, *STIPI* significantly outperformed *A3* for an average of 91.7% across the four time budgets and three sets of test cases.

Notice that 12 comparisons were performed when comparing *STIPI* with each of the five selected approaches (i.e., three test case sets * four time budgets), and thus in total 60 comparisons were conducted. Based on the results, we can observe that *STIPI* significantly outperformed the five selected approaches for 54 out of 60 comparisons (90%), which indicate that *STIPI* has a good capability for solving our test case prioritization problem. In addition, *STIPI* took an average time of 36.5, 51.6 and 82 seconds (secs) for the three sets of test cases. The average running time for the five chosen approaches are: 1) RS: 18, 24.7 and 33.2 secs; 2) *Greedy*: 42, 48 and 54 milliseconds; 3) *A1*: 35.7, 42.8 and 65.5 secs; 4) *A2*: 35.2, 42.2 and 55.4 secs; and 5) *A3*: 8.9, 33.4 and 41.2 secs. Notice that there is no

practical difference in terms of the running time for the approaches except *Greedy*, however the performance of *Greedy* is significantly worse than *STIPI* (Section 5.2), and thus *Greedy* cannot be employed to solve our test case prioritization problem. In addition, based on the domain knowledge of VCS testing, the running time in seconds is acceptable when deployed in practice.

5.3 Overall Discussion

For *RQ1*, we observed that *STIPI* performed significantly better than *RS* for all the *EM*s with the three sets of test cases under the four time budgets. Such an observation reveals that solving our test case prioritization problem is not trivial, which requires an efficient approach. As for *RQ2*, we compared *STIPI* with *Greedy*, *A1*, *A2* and *A3* (Section 4.1). Results show that *STIPI* performed significantly better than *Greedy*. This can be explained that *Greedy* is a local search algorithm that may get stuck in a local space during the search process, while *STIPI* employs mutation operator (Section 4.4) to explore the whole search space towards finding optimal solutions. In addition, *Greedy* converted our multi-objective optimization problem into a single-objective optimization problem by assigning weights to each objective, which may lose many other optimal solutions that hold the same quality [22], while *STIPI* (integrating NSGA-II) produces a set of non-dominated solutions (i.e., solutions with equivalent quality).

When comparing *STIPI* with *A1*, *A2,* and *A3*, the results of *RQ2* showed that *STIPI* performed significantly better than *A1*, *A2,* and *A3* by 83.3% (30/36). Overall *STIPI* outperformed the five selected approaches for 90% (54/60) comparisons. That might be due to two main reasons: 1) *STIPI* considers the coverage of incremental unique elements (e.g., test API commands) when evaluating the prioritization solutions, i.e., only the incremental unique elements covered by a certain test case are taken into account as compared with the already prioritized test cases; and 2) *STIPI* provides the test cases with higher execution positions more influence on the quality of a given prioritization solution. Furthermore, *A2* and *A3* usually work under the assumption that the relations between detected faults and test cases are known beforehand, which is sometimes not the situation in practice, e.g., in our case, we are only aware how many execution times a test case can detect faults rather than having access to the detailed faults detected. However, *STIPI* defined *FDC* to measure the fault detection capability (Section 3.2) without knowing the detailed relations between faults and test cases, which may be applicable to the similar other contexts when the detailed faults cannot be accessed. It is worth mentioning that the current practice of Cisco do not have an efficient approach for test case prioritization, and thus we are working on deploying our approach in their current practice for further strengthening *STIPI*.

5.4 Threats to Validity

The *internal validity* threat arises due to using search algorithms with only one configuration setting for its parameters as we did in our experiment [23]. However, we used the default parameter setting from the literature [24], and based on our previous experience [5, 10], good performance can be achieved for various search algorithms with the default setting. To mitigate the *construct validity* threat, we used the same stopping criteria (50,000 fitness evaluations) for finding the optimal solutions. To avoid *conclusion validity threat* due to the random variations in the search algorithms, we repeated the experiments 10 times to reduce the possibility that the results were obtained by chance. Following the guidelines of reporting the results for randomized algorithms [19], we employed the Vargha and Delaney test as the effect size measure and Mann-Whitney test to determine the statistical significance of results. First *external validity* threat is that one may argue the comparison performed only included RS, *Greedy*, one existing approach and two modified versions of the existing approaches, which may not be sufficient. Notice that we discussed and justified why we chose these approaches in Section 4.1, and it is also possible to compare our approach with other existing approaches, which requires further investigation as the next step. Second *external validity* threat is due to the fact that we only performed the evaluation using one industrial case study. We need to mention that we conducted the experiment using three sets of test cases with four distinct time budgets based on the domain knowledge of VCS testing.

6 Related Work

In the last several decades, test case prioritization has attracted a lot of attention and considerable amount of work has been done [1-3, 8]. Several survey papers [25, 26] present results that compare existing test case prioritization techniques from different aspects, e.g., based on coverage criteria. Followed by the aspects presented in [25], we summarize the related work close to our approach and highlight the key differences from the following three aspects: coverage criteria, search-based prioritization techniques (which is related with our approach) and evaluation metrics.

Coverage Criteria. Existing works defined a number of coverage criteria for evaluating the quality of prioritization solutions [2, 3, 26] such as branch coverage and statement coverage, function coverage and function-level fault exposing potential, block coverage, modified condition/decision coverage, transition coverage and round trip coverage. As compared with the state-of-the-art, we proposed three new coverage criteria driven by the industrial problem (Section 3.2): 1) Configuration coverage (*CC*); 2) Test API coverage (*APIC*) and 3) Status coverage (*SC*).

Search-Based Prioritization Techniques. Search-based techniques have been widely applied for addressing test case prioritization problem [3-5, 10]. For instance, Zhang et al. [3] defined a fitness function with three objectives (i.e., Block, Decision and Statement

Coverage) and integrated the fitness function with hill climbing and GA for test case prioritization. Arrieta et al. [7] proposed to prioritize test cases by defining a two-objective fitness function (i.e., test case execution time and fault detection capability) and evaluated the performance of several search algorithms. The authors of [7] also proposed a strategy to give higher importance to test cases with higher positions (to be executed earlier). A number of research papers have focused on addressing the test case prioritization problem within a limited budget (e.g., time and test resource) using search-based approaches. For instance, Walcott et al. [1] proposed to combine selection (of a subset of test cases) and prioritization (of the selected test cases) for prioritizing test cases within a limited time budget. Different weights are assigned to the selection part and prioritization part when defining the fitness function followed by solving the problem with GA. Wang et al. [5] focused on the test case prioritization within a given limited test resource budget (i.e., hardware, which is different as compared with the time budget used in this work) and defined four cost-effectiveness measures (e.g., test resource usage), and evaluated several search algorithms (e.g., NSGA-II).

As compared with the existing works, our approach (i.e., *STIPI*) defines a fitness function that considers configurations, test APIs and statuses, which were not addressed in the current literature. When defining the fitness function, *STIPI* proposed two strategies, which include 1) only considering the unique elements (e.g., configurations) achieved; and 2) taking the impact of test case execution orders on the quality of prioritization solutions into account, which is not the case in the existing works.

Evaluation Metrics (*EM*s). *APFD* is widely used in the literature as an *EM* [2, 3, 8, 16]. Moreover, the modified version of *APFD* (i.e., $APFD_p$) using time penalty [1, 16] is usually applied for test case prioritization with a time budget. Other metrics were also defined and applied as *EMs* [9, 26] such as Average Severity of Faults Detected, Total Percentage of Faults Detected and Average Percentage of Faults Detected per Cost ($APFD_c$). As compared with the existing *EMs*, we defined in total six new *EM*s driven by our industrial problem for configurations, test APIs, and statuses (Table A-3), which include: 1) *APCC*, *APAC*, and *APSC*, inspired by *APFD*, when there is 100% time budget; and 2) $APCC_p$, $APAC_p$, and $APSC_p$ inspired by $APFD_p$, when there is a limited time budget (e.g., 25% time budget). Furthermore, we defined the seventh *EM* (*MFDC*) to assess to what extent faults can be detected when the time budget is less than 100% (Table A-3). To the best of our knowledge, there is no existing work that applies these seven *EMs* for assessing the quality of test case prioritization solutions.

7 Conclusion and Future Work

Driven by our industrial problem, we proposed a multi-objective search-based test case prioritization approach named *STIPI* for covering maximum number of configurations, test APIs, statuses, and achieving high fault detection capability as quickly as possible. We compared *STIPI* with five test case prioritization approaches using three sets of test cases

with four time budgets. The results show that *STIPI* performed significantly better than the chosen approaches for 90% of the cases. *STIPI* managed to achieve a higher performance than random search for on average 39.9% (configuration coverage), 18.6% (test API coverage), 32.7% (status coverage) and 43.9% (*FDC*). In the future, we plan to compare *STIPI* with more prioritization approaches from the literature using additional case studies with larger scale to further generalize the results.

Acknowledgement

This research is supported by the Research Council of Norway (RCN) funded Certus SFI. Shuai Wang is also supported by the RFF Hovedstaden funded MBE-CR project. Shaukat Ali and Tao Yue are also supported by the RCN funded Zen-Configurator project, the EU Horizon 2020 project funded U-Test, the RFF Hovedstaden funded MBE-CR project and the RCN funded MBT4CPS project.

References

[1] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 1-12. ACM, 2006.

[2] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pp. 179-188. IEEE, 1999.

[3] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *Software Engineering, IEEE Transactions on,* vol. 33, no. 4, pp. 225-237. 2007.

[4] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen, "Multi-objective test prioritization in software product line testing: an industrial case study," in *Proceedings of the 18th International Software Product Line Conference*, pp. 32-41. ACM, 2014.

[5] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-Objective Search," presented at the International Conference on Software Engineering (ICSE), 2016.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* vol. 6, no. 2, pp. 182-197. IEEE, 2002.

[7] A. Arrieta, S. Wang, G. Sagardui, and L. Etxeberria, "Test Case Prioritization of Configurable Cyber-Physical Systems with Weight-Based Search Algorithms," in *Proceedings of the conference on Genetic and evolutionary computation*. ACM, 2016.

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

[9] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 329-338. IEEE Computer Society, 2001.

[10] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "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*, pp. 631-642. ACM, 2016.

[11] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software,* vol. 103, pp. 370-391. 2015.

[12] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 1493-1500. ACM, 2013.

[13] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 619-630. ACM, 2016.

[14] S. Wang, S. Ali, T. Yue, and M. Liaaen, "UPMOA: An improved search algorithm to support user-preference multi-objective optimization," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pp. 393-404. IEEE, 2015.

[15] *Results of the experiments available publicly*. Available: http://www.simula.no/publications/2016-06

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

[17] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham, "AbYSS: Adapting scatter search to multiobjective optimization," *IEEE Transactions on Evolutionary Computation,* vol. 12, no. 4, pp. 439-457. IEEE, 2008.

[18] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132. SAGE journals, 2000.

[19] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 1-10. IEEE, 2011.

[20] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics,* pp. 50-60. 1947.

[21] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software,* vol. 42, no. 10, pp. 760-771. Elsevier, 2011.

[22] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety,* vol. 91, no. 9, pp. 992-1007. Elsevier, 2006.

[23] M. de Oliveira Barros and A. Neto, "Threats to Validity in Search-based Software Engineering Empirical Studies," *UNIRIO-Universidade Federal do Estado do Rio de Janeiro, techreport,* vol. 6, 2011.

[24] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Search Based Software Engineering*, ed: Springer, 2011, pp. 33-47.

[25] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability,* vol. 22, no. 2, pp. 67-120. 2012.

[26] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal,* vol. 21, no. 3, pp. 445-478. 2013.

Paper B

Search-Based Cost-Effective Test Case Selection within a Time Budget: An empirical Study

Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue

Published in Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), July 20-24, 2016.
DOI: 10.1145/2908812.2908850

The layout has been revised.

Abstract

Due to limited time and resources available for execution, test case selection always remains crucial for cost-effective testing. It is even more prominent when test cases require manual steps, e.g., operating physical equipment. Thus, test case selection must consider complicated trade-offs between cost (e.g., execution time) and effectiveness (e.g., fault detection capability). Based on our industrial collaboration within the Maritime domain, we identified a real-world and multi-objective test case selection problem in the context of robustness testing, where test case execution requires human involvement in certain steps, such as turning on the power supply to a device. The high-level goal is to select test cases for execution within a given time budget, where test engineers provide weights for a set of objectives, depending on testing requirements, standards, and regulations.

To address the identified test case selection problem, we defined a fitness function including one cost measure, i.e., Time Difference (TD) and three effectiveness measures, i.e., Mean Priority (MPR), Mean Probability (MPO) and Mean Consequence (MC) that were identified together with test engineers. We further empirically evaluated eight multi-objective search algorithms, which include three weight-based search algorithms (e.g., Alternating Variable Method) and five Pareto-based search algorithms (e.g., Strength Pareto Evolutionary Algorithm 2 (SPEA2)) using two weight assignment strategies (WASs). Notice that Random Search (RS) was used as a comparison baseline. We conducted two sets of empirical evaluations: 1) Using a real world case study that was developed based on our industrial collaboration; 2) Simulating the real world case study to a larger scale to assess the scalability of the search algorithms. Results show that SPEA2 with either of the WASs performed the best for both the studies. Overall, SPEA2 managed to improve on average 32.7%, 39% and 33% in terms of MPR, MPO and MC respectively as compared to RS.

Keywords: Test Case Selection; Search; Multi-Objective Optimization.

1 Introduction

Test case selection is very crucial in practice for cost-effective testing, especially when there is a given budget with limited time and resources that are the bottleneck for exhaustive testing [1]. The work presented in this paper is based on one of the research projects in the Certus Software Verification and Validation Center[1] with an industrial partner from the Maritime domain. The project focuses on developing a new test case execution system for optimizing robustness test execution of their real-time embedded systems deployed in various maritime applications, e.g., dynamic positioning, vessel control, integrated process control.

[1] www.certus-sfi.no

In our industrial context, test cases are created to test a System Under Test (*SUT*). The number of test cases increases over time, and it is practically infeasible to execute all available test cases for testing the *SUT* when a large number of test cases have to be executed manually, such as turning on the power supply to a device and recording observations. Therefore, test case selection is required in the current practice based on various cost (e.g., execution time of test cases) and effectiveness criteria (e.g., fault detection capability). The overall goal of test case selection is to select test cases that can be executed within a time budget, while optimally satisfying various cost and effectiveness objectives. Test engineers provide their preferences of these objectives as weights depending on the testing requirements, standards, and regulations. This type of multi-objective test case selection problem can be formulated as a multi-objective optimization problem [2, 3].

In this paper, we propose a search-based test case selection approach for addressing the above-described problem. The proposed approach takes into account individual objectives with preferences (i.e., weights) and a time budget for executing test cases, which are specified by test engineers based on their preferences [3]. Together with test engineers from our industrial partner, we defined one cost measure, i.e., *Time Difference* (*TD*), which represents difference in period between a time budget and execution cost for a solution, and three effectiveness measures: *Mean Priority* (*MPR*), *Mean Probability* (*MPO*) and *Mean Consequence* (*MC*). Specifically, in our context, each test case has four attributes: *time*, *priority*, *probability*, and *consequence*.

After the test cases have been executed, results of execution are automatically used to update values of the attribute *probability* of test cases. For test case selection, test engineers can select from five weight options of *priority*, three weight options of *probability* and five weight options of *consequence*, based on their preferences. When combining these weight options together, in total, there are 75 possible combinations of weights for test case selection. Notice that we studied all these 75 combinations of weights using two weight assignment strategies (*WAS*s) (i.e., Fixed Weights (*FW*) and Randomly-Assigned Weights (*RAW*).

Furthermore, we conducted an extensive empirical evaluation by evaluating eight existing multi-objective search algorithms (i.e., three weight-based and five Pareto-based algorithms) with the aim to integrate the best algorithm into our test case selection approach. Random Search (RS) is used as a comparison baseline. For the three weight-based search algorithms, we defined a fitness function by taking the four cost/effectiveness measures (e.g., *TD*) and user preferences into consideration. As for the five Pareto-based search algorithms, we incorporated the user preferences as an additional objective in addition to the cost and effectiveness measures.

The empirical evaluation includes: 1) A real world case study that was created based on the real data of our industrial partner; and 2) A set of artificial problems inspired by the real world case study to assess the scalability of the search-based approach. Results show that Strength Pareto Evolutionary Algorithm 2 (SPEA2) with either of the *WAS*s performed

the best among the search algorithms. When compared to RS, SPEA2 achieved on average 32.7%, 39% and 33% better *MPR*, *MPO*, and *MC* respectively. Notice that we did not compare the results with the current practice of the industrial partner in this work, since we are in the process of developing a new test case selection system together with them.

The main contributions of this paper can be summarized as: 1) A multi-objective test case selection problem with user preferences and a time budget is identified and formulated from an industrial maritime domain application through domain analysis; 2) One cost measure and three effectiveness measures are formally defined together with test engineers followed by proposing a fitness function based on the defined cost-effectiveness measures; and 3) A thorough empirical evaluation was performed that includes three weight-based and five Pareto-based search algorithms with two *WAS*s using a real world case study and a large number of artificial problems with RS used as a comparison baseline.

The rest of the paper is organized as follows: Section 2 introduces the selected multi-objective search algorithms and *WAS*s; Section 3 gives a description of the industrial context; and our approach is described in Section 4. Section 5 discusses our experiment design followed by presenting the results in Section 6. Section 7 presents the related work, and the paper is concluded in Section 8.

2 Background

2.1 Multi-Objective Search Algorithms

Weight-based search algorithms combine a multi-objective optimization problem into a single-objective optimization problem with a scalar objective function by assigning weights to each objective [3]. We used three weight-based search algorithms that include one local search algorithm (Alternating Variable Method (AVM) [4]), and two global search algorithms: Genetic Algorithm (GA) [2] and (1+1) Evolutionary Algorithm (EA) [5].

Pareto-based search algorithms are based on the Pareto dominance theory where a set of non-dominated solutions is generated for different objectives. We investigated the existing Pareto-based search algorithms [6, 7], and chose five most-commonly used ones that apply different mechanisms, which includes: Nondominated Sorting Genetic Algorithm (NSGA)-II [8], Multi-Objective Cellular (MOCELL) [9], Strength Pareto Evolutionary Algorithm 2 (SPEA2) [10], CellDE [11] and Indicator Based Evolutionary Algorithm (IBEA) [12] with hypervolume (*HV*) [13] as the performance indicator to evaluate a set of solutions.

2.2 Weight Assignment Strategies

Two weight assignment strategies (*WAS*s) are commonly applied in the existing literature [3], i.e., Fixed Weights (*FW*) and Randomly-Assigned Weights (*RAW*). *FW* assign fixed

normalized weights to each objective based on the domain knowledge (e.g., in the maritime domain, test engineers are more concerned about testing components that have higher chance to cause failure based on earlier testing experience when there is a need to test robustness of the systems) [3]. On the other hand, *RAW* is inspired by Random-Weighted Genetic Algorithm (RWGA), where a randomly-generated set of weights (normalized) to each objective are dynamically assigned at each generation during the search [3]. Notice that the weights generated by *RAW* should satisfy user-defined constraints [3].

3 Industrial Context

We first present the domain analysis conducted to identify key research challenges at our industrial partner's current testing practice in Section 3.1 followed by the results in Section 3.2.

3.1 Domain Analysis

Our industrial partner in this study is one of the leading oil & gas service companies in Norway, which develops general purpose, real -time process control computers used for controlling a wide variety of system application in both on and offshore installations. The department that we are working with focuses on robustness testing, and all their test cases need to be executed manually. Test engineers aim at executing the maximum number of robustness test cases based on different testing criteria (i.e., test important requirements) and within a defined time budget. They are also interested in developing a tool where a test engineer can specify the importance of testing objectives (e.g., detect more faults) for a particular *SUT*, which will then output a list of optimal test cases.

Thus, the problem we identified can be summarized as a multi-objective test case selection problem, i.e., selecting maximum number of test cases for execution from the test suite within the time budget while achieving maximum pre-defined testing criteria. Additionally, we noticed that the test engineers manually add test cases based on different standards (e.g., Norsok standard common requirements U-CR-005), requirements from customers and domain knowledge, and the test cases are usually for system level. They select the test cases manually that is subjective and time consuming process as it relies on domain expertise.

3.2 Results of Domain Analysis

After five meetings and thorough discussions with the test engineers, we learnt that each test case should include four key attributes: 1) *Priority* refers to the importance of a test case that can be determined based on type of requirements under test, e.g., a test case testing safety requirements (e.g., electronics for subsea electronic module can stand a minimum temperature of 70ºC as defined in the standard U-CR-003 has a higher priority

than a test case testing generation of alarm in an optional device); 2) *Probability* indicates the likelihood that a test case may find a fault, which can be calculated based on the execution history of the test case, e.g., a probability with 0.5 indicates out of two executions for a test case, fault was detected one time on average. The first time a test case is added in the repository, this attribute is classified manually based on the domain knowledge of test engineers; 3) *Consequence* determines the impact of a failure of a test case that the system can have on the environment once it is operational (e.g., test cases to check that valves are stable during operation have higher value since their failure can cause huge damage); 4) *Time* specifies the execution time to execute the test case. Notice test engineers enter the values for the attributes (e.g., *priority*) based on the domain knowledge, standards (e.g., U-CR-003), agreed upon customer requirements and user manuals of systems.

4 Search-Based Approach

We represent our search problem (Section 4.1), followed by defining four cost/effectiveness measures (Section 4.2) and fitness function based on these measures (Section 4.3).

4.1 Search Problem Representation

Based on the four identified test case attributes (Section 3.2), we identified four objectives that we classified into one cost and three effectiveness objectives (quantified using respective measures). The cost measure is calculated based on the execution *time* of a test case, and the effectiveness measures are calculated based on *priority*, *probability* and *consequence* of a specific test case.

4.1.1 Basic Concepts

Definition 1. A test suite including a set of test cases is defined as:

$TS = \{tc_1, tc_2, \dots, tc_{nt}\}$, nt is the total number of test cases in TS.

Definition 2. A set of execution time for the test cases in TS is defined as: $ET = \{et_1, et_2, \dots et_{nt}\}$. Any test case tc_i in TS has a corresponding execution *time* in ET i.e., et_i.

Definition 3. A set of priorities for test cases in TS is defined as: $PR = \{pr_1, pr_2, \dots, pr_{nt}\}$. Each tc_i has a corresponding value for *priority* in PR (i.e., pr_i). Each *priority* has a set of five different options that can be configured for each test case with the range of values for the options divided evenly between 0 and 1.

$\forall_{k=1\ to\ nt} pr_k \in \{higher_{pr}, high_{pr}, medium_{pr}, low_{pr}, lower_{pr}\}$, $lower_{pr} = \{0, \dots, 0.2\}$, $low_{pr} = \{0.21, \dots, 0.4\}$, $medium_{pr} = \{0.41, \dots, 0.6\}$, $high_{pr} = \{0.61, \dots, 0.8\}$, $higher_{pr} = \{0.81, \dots, 1\}$.

Definition 4. A set of probabilities for test cases in TS is stated as: $PO = \{po_1, po_2, \dots, po_{nt}\}$. Each tc_i has a corresponding value for *probability* in PO (i.e., po_i). Each *probability* has three different options for each test case with range of values divided equally, i.e., $\forall_{k=1\ to\ nt} po_k \in \{high_{po}, medium_{po}, low_{po}\}$, $low_{po} = \{0, \dots, 0.33\}$, $medium_{po} = \{0.34, \dots, 0.66\}$, $high_{po} = \{0.67, \dots, 1\}$.

Definition 5. A set of consequences for a test case in TS is expressed as: $C = \{c_1, c_2, \dots, c_{nt}\}$. Each tc_ihas a corresponding value of *consequence* in C (i.e., c_i). Each *consequence* has five different options for selection for each test case with the range of values divided equally across different options:
$\forall_{k=1\ to\ nt} c_k \in \{higher_c, high_c, medium_c, low_c, lower_c\}$, $lower_c = \{0, \dots, 0.2\}$, $low_c = \{0.21, \dots, 0.4\}$, $medium_c = \{0.41, \dots, 0.6\}$, $high_c = \{0.61, \dots, 0.8\}$, $higher_c = \{0.81, \dots, 1\}$.

Definition 6. A set of cost measures is defined as:

$$Cost = \{cost_1, cost_2, \dots, cost_{ncost}\}$$

Definition 7. A set of effectiveness measures is defined as:

$$Effect = \{effect_1, effect_2, \dots, effect_{neffect}\}$$

Definition 8. A set of potential solutions is represented as:
$S = \{s_1, s_2, \dots, s_{ns}\}$, where each solution s_j consists of a set of selected test cases from test suite TS for testing a particular System Under Test within a given time budget (cost measure) while achieving maximum for effectiveness measures (e.g., *priority*).

4.1.2 Problem Representation

Based on the above-mentioned concepts, our test case selection problem can be formulated as: Search a solution s_k from the total number of ns solutions in S to obtain highest effectiveness while having the cost as close to the time budget (i.e., tb) as possible, such that the cost can not exceed the time budget:

$$\forall_{i=1\ to\ neffect} \forall_{j=1\ to\ ns}\ Effect\ (s_k, effect_i) \geq Effect\ (s_j, effect_i)$$

$$\forall_{i=1\ to\ ncost} \forall_{j=1\ to\ ns}\ (Cost\ (s_k, cost_i) \leq tb) \cap ((tb - Cost\ (s_k, cost_i)) \leq (tb - Cost\ (s_j, cost_i)))$$

For a solution s_j, $Effect\ (s_j, effect_i)$ returns the i_{th}effectiveness and $Cost\ (s_j, cost_i)$ returns the i_{th}cost.

4.2 Cost/Effectiveness Measures

Cost Measures *Cost* consists of one element in our context, i.e., time difference (*TD*), which is a difference in period between a time budget (tb) and execution cost (*EC*) for a solution s_j.

Measure Formula 1. The execution cost (EC) is calculated with: $EC_{s_j} = \sum_{i=1}^{\mathrm{nj}} et_{ji}$, where $1 \leq nj \leq nt$ and $et_{ji} \in ET_j$.

Measure Formula 2. The *Time Difference* (TD) for a solution s_j is measured as: $TD_{s_j} = tb - EC_{s_j}$.

Effectiveness Measures *Effect* consists of three elements: *Mean Priority* (*MPR*), *Mean Probability* (*MPO*) and *Mean Consequence* (*MC*). *Priority* and *consequence* for each test case tc_i has one value out of five different values (from Section 4.1.1). Test engineers can manually change these values. However, the *probability* of test case is calculated dynamically based on its execution history, for a test case tc_i:

$$Failure\ Rate_{tc_i} = \frac{Number\ of\ times\ tc_i\ found\ a\ fault}{Number\ of\ times\ tc_i\ was\ executed}$$

The *Failure Rate* of a test case is used to indicate options for *probability* of a test case using probability set (*PO*) as described in Section 4.1.1. For a solution s_j with *priority* (*pr*), *probability* (*po*) and *consequence* (*c*): *MPR*, *MPO* and *MC* can be represented as:

Measure Formula 3. *MPR* is the mean *priority* that measures the average priority for the optimized test cases in a solution, and can be calculated as: $MPR_{s_j} = \frac{\sum_{i=1}^{nj} pr_{ji}}{nj}$, where $pr_{ji} \in PR_j$.

Measure Formula 4. *MPO* is the mean *probability* of all the test cases in a solution such that: $MPO_{s_j} = \frac{\sum_{i=1}^{nj} po_{ji}}{nj}$, where $po_{ji} \in PO_j$.

Measure Formula 5. *MC* is the mean *consequence* of all the test cases in a solution expressed as: $MC_{s_j} = \frac{\sum_{i=1}^{nj} c_{ji}}{nj}$, where $c_{ji} \in C_j$.

All the effectiveness measures range from 0 to 1 with a higher value representing a better solution.

4.3 Fitness Function

As described in Section 2, weight-based and Pareto-based search algorithms have different working mechanisms. Since weight-based search algorithms convert a multi-objective problem into a single-objective problem, we can compare the values for fitness function to measure their performance [3]. However, Pareto-based search algorithms only require defining a set of objective functions (i.e., cost/effectiveness measures) without converting all the objective functions into one [3]. Thus, we define the fitness functions separately for them as below.

4.3.1 For Weight-Based Search Algorithms

Since the values obtained by different cost/effectiveness measures may not be comparable, we normalize the values obtained by *TD* using the normalization function suggested in [14] to obtain values in the magnitude from 0 to 1: $Nor\left(F(x)\right) = \frac{F_x}{F_x+1}$. Notice that test engineers have user preferences for different cost and effectiveness measures (i.e., *MPR*, *MPO*, *MC*, *TD*) that are represented by assigning weights to each objective (i.e., w_{pr}, w_{po}, w_c, w_t) in our context. For each w_{pr} and w_c, the test engineer selects one option out of the five (i.e., *Higher*, *High*, *Medium*, *Low*, *Lower*), and for w_{po} one out of three different options (i.e., *High*, *Medium*, *Low*). As for w_t, test engineers can specify a fixed value beforehand based on the domain knowledge, e.g., 0.2 in our context.

The fitness function for weight-based search algorithms can be calculated by: $FitnessFun(FF) = 1 - \left(MPR \times w'_{pr} + MPO \times w'_{po} + MC \times w'_c + \left(1 - Nor(TD)\right) \times w'_t\right)$. w'_{pr}, w'_{po}, w'_c and w'_t are the weights that will be used during the search process calculated as:

$$Total = w_{pr} + w_{po} + w_c, w_t = w'_t, w'_{pr} = \frac{w_{pr}}{Total} \times (1 - w_t)$$

$$w'_{po} = \frac{w_{po}}{Total} \times (1 - w_t), w'_c = \frac{w_c}{Total} \times (1 - w_t)$$

Notice that w'_{pr}, w'_{po}, w'_c and w'_t should satisfy: $w'_{pr} + w'_{po} + w'_c + w'_t = 1$, and we subtracted 1 in FF to imply that a value closer to 0 is better.

4.3.2 For Pareto-Based Search Algorithms

We select the cost measure (i.e., *TD*) after normalization: $Nor\left(F(x)\right) = \frac{F_x}{F_x+1}$, and the effectiveness measures (i.e., *MPR*, *MPO*, *MC*) as objective functions for the Pareto-based search algorithms after subtracting 1 in $MPR_{s_j}, MPO_{s_j}, MC_{s_j}$ to ensure that solution with values closer to 0 is better. Notice that Pareto-based search algorithms usually treat all the objectives equally [3], and we use the above-defined fitness function as the fifth objective for the selected Pareto-based algorithms to reflect user preferences. Therefore, for all the Pareto-based search algorithms, the above-defined five objective functions are applied to guide the search to find optimal solutions.

5 Experiment Design

In this section, we present the experiment design (as shown in Table B-1), which includes: 1) research questions (Section 5.1); 2) real world case study and artificial problems (Section 5.2). After that, we provide the evaluation metrics for the experiments (Section 5.3), describe our experiment design (Section 5.4), and present the parameter settings used in the algorithms (Section 5.5).

5.1 Research Questions

RQ1: Are weight-based and Pareto-based search algorithms effective to solve our selection problem as compared with RS?

RQ2: Among weight-based search algorithms with the two weight assignment strategies (*WAS*s), which one can achieve the best performance in terms of solving our test case selection problem?

RQ3: For Pareto-based search algorithms with the two *WAS*s, which one can perform the best for our test case selection problem?

RQ4: Between the best weight-based search algorithm and the best Pareto-based search algorithm, which one is better?

RQ5: Are weight-based and Pareto-based search algorithms scalable for solving test case selection problems of varying complexity using the different *WAS*s?

Table B-1. An Overview of the Experiment Design*

RQ	T	Description	Algorithms	*WAS*	Evaluation Metric	Statistical Tests	Study
1	T_1	Comparison of algorithms^{wb} with RS using each *WAS*	3 Weight-based	*FW* *RAW*	Fitness values	Vargha and Delaney $\hat{A}_{12}$, Mann-Whitney U test	CS_1 CS_2
	T_2	Comparison of algorithms^{pb} with RS using each *WAS*	5 Pareto-based		*HV*		
2	T_3	Comparison of algorithms^{wb} with each other using both *WAS*s	3Weight-based		Fitness values		
3	T_4	Comparison of algorithms^{pb} with each other using both *WAS*s	5 Pareto-based		*HV*		
4	T_5	Comparison of the best weight based and Pareto-based search algorithm	1Weight-based 1 Pareto-based	n/a	*HV*		
5	T_6	Evaluation of the scalability of algorithms^{wb}	3 Weight-based	*FW* *RAW*	Mean Fitness Value (MFV_i)	Kendall's Tau (τ)	CS_2
	T_7	Evaluation of the scalability of algorithms^{pb}	5 Pareto-based		Mean *HV* (MHV_i)		

*T: Task, ^{wb}: Weight-based, ^{pb}: Pareto-based.

5.2 Case Study and Artificial Problems

Real World Case Study: Due to confidentiality issues with industrial data, we created a real world case study with high-level test cases for the key elements of Subsea Oil and Gas Production System based on the key attributes of test cases used at the industrial partner (Section 3). We used different standards (e.g., Design and operation of subsea production systems - ISO 13628-6:2006, Norsok standard common requirements U-CR-005, U-CR-006Rev.1, U-CR-003), requirements from oil and gas companies available publicly in the internet; and domain knowledge to categorize the attributes *priority* and *consequence*, and fill execution time for each test case.

Furthermore, to obtain the failure information of each test case, we checked different subsea components from OREDA Offshore Reliability Data Handbook [15]. Specifically, we investigated the mean failure rate (per 10^6) of the different components (e.g., sensors, valves), and distributed them equally across different options as defined in Section 4.1.1.

Based on the standards, public requirements and handbook, we created a real world case study (CS_1) that includes 165 test cases, and each test case has the four key attributes (i.e., *priority*, *probability, consequence, time*). For our experiments, we map w_{pr}, w_{po} and w_c with different options of pr_k, po_k and c_k respectively. Recall (from Section 4.1.1) that pr_k has five options, po_k has three options and c_k has five options. Note that we tried all the combinations of weights that a test engineer could assign when trying to optimize the tests for different objectives. Therefore, for a fixed number of test cases with specific value for w_t (i.e., weight for *TD*), in total 75 problems were defined.

We compare two *WAS*s (i.e., *FW*, *RAW*) for assigning the weights of objectives. For *FW*, we assign a fixed weight for the different objectives: $pr_k \in \{1,0.8,0.6,0.4,0.2\}$, $po_k \in \{1,0.66,0.33\}$ and $c_k \in \{1,0.8,0.6,0.4,0.2\}$, while for *RAW* we assign a random weight at each generation between the ranges of categories (from Section 4.1) for each problem. For example, for a problem with $pr_i = higher$, $po_i = low$ and $c_i = lower$ for *FW:* $w_{pr_{FW}} = 1, w_{po_{FW}} = 0.33, w_{c_{FW}} = 0.2$; for *RAW:* $w_{pr_{RAW}}, w_{po_{RAW}}$ and $w_{c_{RAW}}$ is random between $0.81, \dots, 1$, $0, \dots, 0.33$, and $0, \dots, 0.2$ respectively for each generation. After that we normalize the values of the weights (e.g., w_{pr}) as in Section 4.3. We take 30% of the total test case execution time as the budget for test case selection to calculate cost measure (*TD*), since our industrial partner typically uses similar execution time for test case selection. For all our experiments the weight for time (w_t) is set constant as 0.2 based on our discussion with the test engineers at the industrial partner, i.e., $w_t = 0.2$ for *TD* in Section 4.3. Thus, $w'_{pr} + w'_{po} + w'_c = 1 - 0.2 = 0.8$.

Artificial Problems: Based on the real world case study (CS_1), we created artificial problems (CS_2). CS_2 consists of ten test suites (TSs) with the number of test cases ranging from 100 to 1000 with an increment of 100. The test cases are all randomly generated, such that the attributes (*priority*, *probability* and *consequence*) are distributed uniformly as defined in Section 4.1 and have execution *time* in the same range as in the real world case study (0.5 to 8 hours for each test case). We take 30% of total test execution time as the time budget in our experiments. We have a total of 750 problems for CS_2, since there are 75 problems for one TS.

5.3 Evaluation Metrics

To address *RQ1* and *RQ2*, we use the fitness values produced by them (using the fitness function in Section 4.3). We evaluate the three weight-based search algorithms (i.e., AVM, GA, (1+1) EA) and RS as shown in Table B-1. Similarly, to address *RQ1* and *RQ3*, we employ a commonly-used quality indicator called hypervolume (*HV*) [13, 16]. We evaluate the overall performance of the five Pareto-based search algorithms (i.e., NSGA-II, SPEA2, CELLDE, IBEA, MOCELL) and RS (as shown in Table B-1). A higher value of *HV* indicates a better performance of the algorithm. Moreover, we use *HV* to evaluate the best

weight-based and Pareto-based search algorithms for addressing *RQ4* as depicted in Table B-1. To address *RQ5*, we used two evaluation metrics as presented in Table B-1.

1) Mean fitness value of a test suite (*TS*) i used for the weight-based search algorithms is calculated by: $\forall_{i=1\ to\ 10}\ MFV_i = \frac{\frac{\sum_{j=1}^{75} AFV_{ij_{FW}}}{75} + \frac{\sum_{j=1}^{75} AFV_{ij_{RAW}}}{75}}{2}$, where $AFV_{ij_{FW}}$ and $AFV_{ij_{RAW}}$ represent the average fitness values for TS_i for the j_{th} problem with *FW* and *RAW* respectively.

2) Mean *HV* of a test suite (*TS*) i used for the Pareto-based search algorithms is computed by: $\forall_{i=1\ to\ 10}\ MHV_i = \frac{\frac{\sum_{j=1}^{75} AHV_{ij_{FW}}}{75} + \frac{\sum_{j=1}^{75} AHV_{ij_{RAW}}}{75}}{2}$, where $AHV_{ij_{FW}}$ and $AHV_{ij_{RAW}}$ represent the average *HV* for TS_i for the j_{th} problem with *FW* and *RAW* respectively. We combined the different *WAS*s to a single metric (i.e., *MFV* and *MHV*) with the aim to evaluate the scalability of the algorithms.

5.4 Experiment Tasks

We used seven experiment tasks (T_1 - T_7) for answering *RQ1* - *RQ5* as presented in Table B-1. For example, to answer *RQ1*, we used T_1 and T_2 as shown in Table B-1. T_1 is performed to compare three weight-based search algorithms with RS using each *WAS* (i.e., *FW* and *RAW*) by comparing the fitness values (from the fitness function in Section 5.3) as the evaluation metric using two statistical tests (Vargha and Delaney $\hat{A}_{12}$ that is a non-parametric effect size measure [17] and Mann-Whitney U test that tells whether results are statistically significant [18] based on the guidelines for statistical tests for randomized algorithms [19]) for both real world case study (CS_1) with 75 problems and artificial problems (CS_2) consisting of 750 problems. In T_2, we compared five Pareto-based search algorithms using *HV*.

5.5 Parameter Settings

Notice that the selected search algorithms together with the quality indicator (i.e., *HV*) were implemented based on jMetal, which is a Java-based framework [20]. We have used similar parameter settings for all the algorithms to maintain consistency across them, and as used in jMetal [20] except with fitness evaluations of 20,000. Moreover, we executed each problem 100 times to account for random variation in the algorithms. We implemented a random number generator [21] in Java for *RAW*. All the experiments were executed on the Abel cluster at the University of Oslo (UiO) [22].

6 Results and Analysis

6.1. Real World Case Study

RQ1: Recall that *RQ1* is designed to check whether search algorithms are effective to solve our test case selection problem. This is done by checking whether the search algorithms perform better than RS. Using the Vargha and Delaney Statistics and the Mann Whitney U Test to analyse the results, we noticed that all the search algorithms performed significantly better than RS for all the problems, i.e., $\hat{A}_{12}$ for the search algorithms is greater than 0.5, and *p*-value is less than 0.05. Detailed results are available in [21].

RQ2: This question is designed to evaluate the three weight-based search algorithms (i.e., AVM, GA, (1+1) EA) with the two *WAS*s (i.e., *FW*, *RAW*). We can analyse the results as shown in Fig. B-1. We split two algorithms *A* and *B* in Fig. B-1 to show results in regard to the two statistical tests (i.e., Vargha and Delaney Statistics and the Mann Whitney U Test). For example, AVM vs GA in Fig. B-1 implies *A*=AVM and *B*=GA. *A>B* means the number of problems out of 75 for which AVM has significantly better performance than GA ($\hat{A}_{12} > 0.5 \;\&\&\; p < 0.05$), *A<B* means the opposite ($\hat{A}_{12} < 0.5 \;\&\&\; p < 0.05$), and *A=B* means the number of problems for which there is no significant difference in performance between AVM and GA ($p \geq 0.05$).

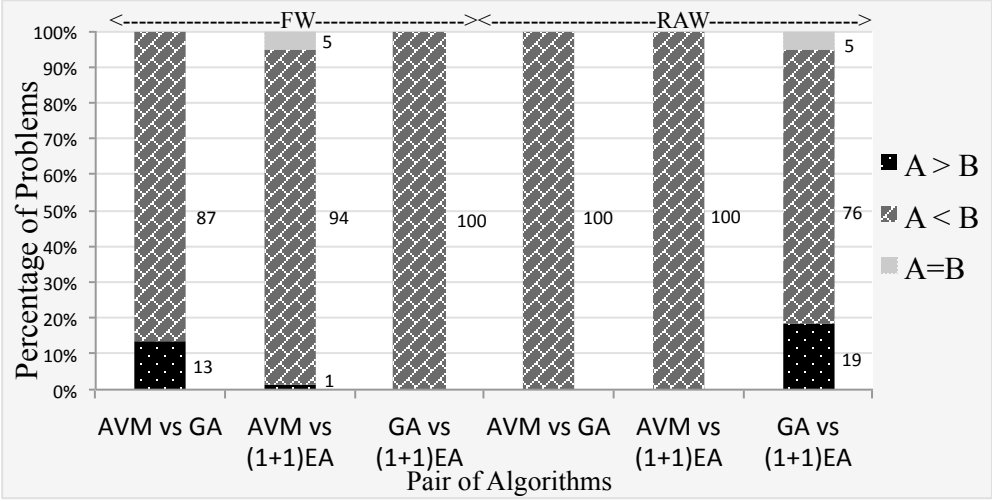

Fig. B-1. Results for RQ2 using *FW* and *RW*

Based on Fig. B-1, we can observe that (1+1) EA performed the best for both the *WASs*. When comparing the two *WAS*s, (1+1) EA using *FW* (i.e., (1+1) EA_FW) performed significantly better than (1+1) EA using *RAW* for 95% (71/75) of the total problems.

RQ3: *RQ3* is designed to evaluate the selected five Pareto-based search algorithms (i.e., NSGA-II, SPEA2, CELLDE, IBEA, MOCELL) along with the two *WAS*s for finding out which algorithm with which *WAS* performs the best. To summarize the key results, Table B-2 gives the percentage of problems where a Pareto-based search algorithm significantly outperforms (i.e., $\widehat{\mathrm{A}}_{12} > 0.5 \;\&\&\; \mathrm{p} < 0.05$) all the other Pareto-based search algorithms

while using *FW* and *RAW*, respectively. Table B-2 also shows the results of comparison between the best Pareto-based search algorithms from the two *WAS*s. Note that sum of percentage is less than or equal to 100%, since some of the results are not significant (i.e., $p \geq 0.05$). More detailed results are in [21].

Table B-2. Percentage of Problems where One Pareto-based Search Algorithm Significantly Outperforms the Others*

WAS	Pareto-Based Algorithm					Best A_p (A_{FW} vs A_{RAW})
	A_1	A_2	A_3	A_4	A_5	
FW	0.0	12.0	0.0	0.0	0.0	A_1_FW
RAW	0.0	100	0.0	0.0	0.0	(65.3)

*A_1: NSGA-II, A_2: SPEA2, A_3: CELLDE, A_4: IBEA, A_5: MOCELL, A_p: Pareto-based algorithm, A_{FW}: Best Pareto-based algorithm using *FW*, A_{RAW}: Best Pareto-based algorithm using *RAW*.

From Table B-2, we can observe that SPEA2 performed the best for both *WASs*. SPEA2 performed the best for 12% and 100% of the total problems using *FW* and *RAW* respectively. Furthermore, SPEA2 with *FW* (SPEA2_FW) performed better than SPEA2 with *RAW* for 65.3% of the problems.

RQ4: Recall that *RQ4* is designed to compare the best weight-based search algorithm with the best Pareto-based search algorithm. With this aim, we compared (1+1) EA_FW (Fig. B-1) with SPEA2_FW (Table B-2); and the results show that SPEA2_FW performed significantly better for all the problems.

The mean execution time of all the three weight-based search algorithms were between 150 to 450 milliseconds (ms) to get a single solution with AVM taking the least amount of time (i.e., 150 ms). Similarly, five Pareto-based search algorithms took 750 to 4,000 ms with NSGA-II taking the least time (i.e., 796 ms) and IBEA taking the most time (i.e., 3961 ms). However, it is worth mentioning that the proposed search-based approach is an offline-based solution for test case selection in our industrial context, and the execution time of the search algorithms is not a critical factor, since there is no large difference between them.

Concluding Remarks: Based on the results, we can conclude that all the weight-based and Pareto-based search algorithms are effective to solve our test case selection problem (*RQ1*). For the weight-based search algorithms, (1+1) EA with *FW* achieved the best performance (*RQ2*), while for the Pareto-based search algorithms SPEA2 with *FW* performed the best although the results are not significant (*RQ3*). Finally, SPEA2 with *FW* significantly outperformed the best weight-based search algorithm (*RQ4*).

6.2. Artificial Problems

RQ1: All the weight-based search algorithms using *FW* significantly outperformed RS for all the problems, while weight-based search algorithms using *RAW* significantly outperformed RS for most of the problems (more than 65%). On the other hand, all the Pareto-based search algorithms with both the *WAS*s significantly outperformed RS for all the problems. Complete results are at [21].

RQ2: Table B-3 presents summarized results of the percentage of problems where a specific algorithm significantly outperforms the other algorithms for ten different test suites (each test suite has 75 problems) similar to Table B-2.

Table B-3. Percentage of Problems where One Pareto-based Search Algorithm Significantly Outperforms the Others*

# Test Cases	*FW*			*RAW*			Best A_w (A_{FW} vs A_{RAW})
	A_1	A_2	A_3	A_1	A_2	A_3	
100	0.0	0.0	100	0.0	12.0	85.3	A_3_FW(100.0)
200	13.3	0.0	86.7	0.0	36.0	64.0	A_3_FW(100.0)
300	6.7	0.0	93.3	0.0	88.0	9.3	A_3_FW(100.0)
400	13.3	0.0	86.7	0.0	96.0	2.7	A_3_FW(100.0)
500	100.0	0.0	0.0	1.0	93.3	4.0	A_1_FW(100.0)
600	100.0	0.0	0.0	0.0	93.3	6.7	A_1_FW(100.0)
700	100.0	0.0	0.0	0.0	93.3	5.3	A_1_FW(100.0)
800	100.0	0.0	0.0	0.0	94.7	5.3	A_1_FW(100.0)
900	100.0	0.0	0.0	0.0	94.7	5.3	A_1_FW(100.0)
1000	100.0	0.0	0.0	0.0	97.3	2.7	A_1_FW(100.0)

*A_1: AVM, A_2: GA, A_3: (1+1) EA, A_w: Weight-based algorithm, A_{FW}: Weight-based algorithm using *FW*, A_{RAW}: Weight-based algorithm using *RAW*.

For *RQ2*, we can observe from Table B-3 that (1+1) EA with *FW* performed the best for lower number of test cases (<=400), while AVM achieved the best performance when the size of test suite grows (>=500). On the other hand for *RAW*, (1+1) EA performed the best for lower number of test cases (<=200), and GA outperformed the other weight-based search algorithms for 300 and more number of test cases. Furthermore, Table B-3 also shows the results of comparing the best algorithms using *FW* and *RAW*, respectively. We can conclude *RQ2* as: 1) For *FW*: a) (1+1) EA performed the best when the number of test cases is equal or less than 400; b) AVM achieved the best performance when the size of test suite is equal or more than 500, and 2) For *RAW*: a) (1+1) EA had the best performance for 200 or lower number of test cases; b) GA achieved the best results for 300 and more test cases.

Table B-4. Percentage of Problems where One Pareto-based Search Algorithm Significantly Outperforms the Others*

#Test Cases	*FW*					*RAW*				
	A_1	A_2	A_3	A_4	A_5	A_1	A_2	A_3	A_4	A_5
100	0.0	100.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
200	0.0	93.3	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
300	0.0	69.3	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
400	0.0	32.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
500	0.0	12.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
600	0.0	8.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
700	0.0	22.7	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
800	0.0	18.7	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
900	0.0	45.3	0.0	0.0	1.3	0.0	100.0	0.0	0.0	0.0
1000	0.0	52.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0

RQ3: Table B-4 presents the summarized results of percentage of problems where an algorithm significantly outperformed all the other algorithms for both the *WAS*s. Based on

Table B-4, we can see that SPEA2 performs the best among all the search algorithms using *FW* and *RAW*. However, when comparing the best algorithm using *FW* and *RAW*, the performance is not consistent; SPEA2 using *FW* performed the best (e.g., # of test cases =100, 200, 400, 600), and SPEA2 using *RAW* performed the best (e.g., # of test cases =300, 500, 700, 800, 1000). For 900 test cases, there was no significant difference between *FW* and *RAW*.

RQ4: The results ([21]) show that the best Pareto-based search algorithm (i.e., SPEA2) significantly outperformed the best weight-based search algorithm (i.e., (1+1) EA or AVM) for more than 90% of the total problems for all the test suites.

RQ5: For the weight-based search algorithms, we chose the mean fitness value (*MFV*) (Section 5.3) of all 75 problems in a test suite, and test the relationship between the *MFV* and number of test cases (increasing in a constant rate) with 10 test suites using Kendall's Tau (τ) in Table B-5. Values of τ range from -1 (i.e., strong negative correlation) to +1 (i.e., strong positive correlation), while a value of zero indicates that there is no correlation. Additionally, we use Prob>|ρ| to report significance of correlation, where a value less than 0.05 means that correlation is statistically significant.

Table B-5. Kendall's Correlation Analysis between *MFV* and Test Cases

Algorithm	***FW***		***RAW***	
	τ	**Prob>|ρ|**	**τ**	**Prob>|ρ|**
AVM	-0.4667	0.0603	0.8222	**0.0009**
GA	0.6000	**0.0157**	0.9556	**0.0001**
(1+1) EA	0.2444	0.3252	0.9111	**0.0002**

From Table B-5, we can observe that 1) For *FW*: there is a negative correlation between *MFV* and number of test cases for AVM (i.e., performance improves on increasing the number of test cases), however, the result is not significant. As for GA, there is a significantly positive correlation between *MFV* and number of test cases that indicates the performance of GA significantly decreases as increasing the number of test cases. For (1+1) EA, there is a positive correlation but not statistically significant; 2) For *RAW*: we can see that there is a significantly positive correlation between *MFV* and number of test cases for AVM, GA and (1+1) EA.

For the Pareto-based search algorithms, we take the mean *HV* (Section 5.3) of all 75 problems in a test suite, and analyse the relationship between the mean *HV* and number of test cases using Kendall's Tau (τ). From Table B-6, we can observe that there is a significantly negative correlation between mean *HV* and number of test cases, i.e., the performance decreases as the number of test cases grows for all the Pareto-based search algorithms, while using both the *WAS*s except for IBEA with *FW* where it is not major.

Table B-6. Kendall's Correlation Analysis between *HV* and Test Cases

Pareto-based Algorithm	*FW*		*RAW*	
	τ	**Prob>|ρ|**	**τ**	**Prob>|ρ|**
NSGA-II	-0.9556	**0.0001**	-0.9556	**0.0001**
SPEA2	-0.9556	**0.0001**	-0.9556	**0.0001**
CELLDE	-1.0000	**<0.0001**	-1.0000	**<0.0001**
IBEA	0.4222	0.0892	-0.9556	**0.0001**
MOCELL	-0.8667	**0.0005**	-0.9111	**0.0002**

Concluding Remarks: Based on the results, all the weight-based and Pareto-based search algorithms are effective to solve our test case selection problem. For *RQ2*, (1+1) EA using *FW* performed the best for lower number of test cases (i.e., 400 or less), while AVM using *FW* performed better for higher number of test cases (i.e., 500 or more). Similarly, for *RQ3* and *RQ4*, we observed that SPEA2 using *FW* and *RAW* performed the best for different test suites among the eight search algorithms. The results for *RQ1–RQ4* are consistent with the findings for the real world case study. For RQ5, the performance of all the search algorithms using the two *WAS*s, i.e., *FW* and *RAW* (except AVM, (1+1) EA, and IBEA using *FW*) seem to decrease significantly with the increase in the number of test cases. However, the result is not significant for the remaining algorithms (i.e., AVM, (1+1) EA, and IBEA using *FW*).

6.3. Overall Discussion

All the three weight-based and five Pareto-based search algorithms with both the weight assignment strategies (i.e., *FW* and *RAW*) for CS_1 and CS_2 significantly outperformed RS (*RQ1*), which means that our test case selection problem is complex to solve. Regarding *RQ2*, using *FW* (1+1) EA performed the best for lower number of test cases (<=400) and AVM performed the best for higher number of test cases (>=500) due to the reason that with a smaller search space, the global search algorithms with a sufficient number of fitness evaluations (i.e., 20,000) manage to find a better solution by exploring more search space as compared to the local search algorithm. When the problems become more complex (>= 500), it may require more fitness evaluations for global search algorithms to find optimal solutions, which will be investigated in the future work. SPEA2 performed the best as compared with all the algorithms for the real world case study (75 problems) and 750 artificial problems out of all Pareto-based search algorithms (*RQ3*) and weight-based search algorithms (*RQ4*). It might be explained as: SPEA2 provides an increasingly more diversity of solutions as the number of objectives are increased (greater than 2), which has been discussed in [10].

For the weight-based search algorithms (RQ2), *FW* performed significantly better than *RAW*, whereas for the Pareto-based search algorithms there was no difference in the performance using either of them. This can be explained due to the fact that weight-based search algorithms converts a multi-objective optimization algorithm into a single-objective algorithm by assigning weights to each objective, where weights play a primary role for the performance of the weight-based search algorithms. In addition, the weights provided

by test engineers (*FW*) in our case are more accurate to guide search towards finding optimal solutions as compared with the randomly generated weights (*RAW*). However, for Pareto-based search algorithms, the weights are only embedded into one of the five objectives, which may not have a major impact on the performance of search algorithms.

Table B-7. Average percentage for each objective where an algorithm is better than RS for both the studies*

Algorithms	*WAS*	CS_1			CS_2		
		Obj 1	**Obj 2**	**Obj 3**	**Obj 1**	**Obj 2**	**Obj 3**
AVM	*FW*	28.5	-4.3	33.3	3.2	11.3	58.9
	RAW	13.0	0.9	13.4	6.8	6.4	6.6
GA	*FW*	35.2	1.8	44.3	2.9	9.7	50.4
	RAW	27.2	3.4	27.6	17.6	17.5	18.0
(1+1) EA	*FW*	39.8	9.4	50.6	6.9	17.0	58.2
	RAW	40.2	6.0	52.2	12.0	12.1	12.4
NSGA-II	*FW*	9.2	43.5	14.8	27.6	30.9	20.6
	RAW	11.3	40.5	18.8	24.2	28.5	23.6
SPEA2	*FW*	26.9	38.4	31.3	38.7	43.0	31.0
	RAW	29.9	33.4	35.6	35.1	41.0	34.1
CELLDE	*FW*	8.5	22.5	9.6	9.3	9.3	-0.1
	RAW	11.6	18.4	14.8	4.2	5.5	4.3
IBEA	*FW*	51.1	19.1	58.2	36.4	39.5	34.9
	RAW	50.1	17.2	57.2	35.4	38.3	33.9
MOCELL	*FW*	19.7	31.8	25.8	24.3	28.2	24.1
	RAW	17.6	35.4	21.9	27.6	30.7	20.0

* Obj 1: *MPR*, Obj 2: *MPO*, Obj 3: *MC*.

Therefore, even test engineers provided an accurate set of weights, it is still possible that *FW* did not significantly outperform *RAW*. Furthermore, we studied to what extent the search algorithms can improve the effectiveness as compared with the random search. Table B-7 shows the average percentage by which an algorithm is able to improve the three effectiveness measures (objectives) using two different *WAS*s with respect to the random search. Recall that for the cost measure, we take time difference (*TD*) to select maximum number of test cases (Section 4.2), and the results show that for all the artificial problems, AVM and (1+1) EA using *FW* were able to select test cases with execution time exactly equal to the time budget, while the selected test cases by other algorithms could not manage to use up the exact time budget.

From Table B-7 we can see that for the real world case study (CS_1), SPEA2 using *FW* performed better on average for 26.9%, 38.4% and 31.3% for the three objectives. Similarly, while using *RAW:* SPEA2 had an average better performance for 29.9%, 33.4% and 35.6% respectively for the three objectives. Likewise for CS_2, SPEA2 performed better for 38.7%, 43.0% and 31.0% using *FW*; and 35.1%, 41.0% and 34.1% using *RAW* for the three objectives. For the overall improvement of SPEA2 as compared to RS, we calculated the average improvement for each objective. The performance of all the eight search algorithms decreased as the increasing number of test cases. This could be due to the reason that as the search space increases, the problem becomes more complex, and thus, it

is difficult to find better solution. However, note that the search algorithms were still much more efficient than the random search.

6.4. Threats to Validity

To reduce the *construct validity* threat, we used the same stopping criteria (20,000 fitness evaluations) to find the optimal solutions. Experiments with only one-default configuration settings for the parameters of the selected search algorithms is a potential threat for *internal validity* [23]. However, these settings conform with the common guidelines in the literature [24]. Moreover, to avoid *conclusion validity* due to the random variations, we repeated our experiments 100 times to reduce the probability of obtaining the results by chance. Additionally, we used the Vargha and Delaney test as the effect size measure and Mann-Whitney test to determine the statistical significance of results as suggested in the guidelines [19]. We used Kendall's tau to measure the impact of increase in test case number on the performance of the algorithms since it is primarily used for non-parametric correlation [25]. It is worth mentioning that we did not compare our solution with the current practice in the industry since we are developing a new test case selection system together with test engineers.

7 Related Work

The work in [26] utilizes information from the previous test cycles (i.e., historical execution data) to select a subset of test cases for execution using regression test selection strategies (e.g., random/ad-hoc technique, minimization technique), such that test cases that have not been executed recently are assigned higher probabilities for test selection. However, all our cost/effectiveness measures are different compared to [26]. In [27], an approach was proposed where a two-objective problem (i.e., code coverage, execution time) was converted into a single-objective problem using weights for the fitness function. One objective was similar to our cost measure – *Time Difference* (*TD*), i.e., they also consider the testing time budget. However, it is different from our work since it focuses on the code level of the system (i.e., code coverage defined in [27]) unlike our industrial case study where the testing is focused on the system level (i.e., our effectiveness measures are also different). The authors in [28] selected test cases for problems with two objectives (i.e., code coverage and execution time) and three objectives with an addition of fault history by using two Pareto-based search algorithms (NSGA-II, and its variation: vNSGA-II), and one weight-based search algorithm (i.e., Greedy). We use two similar objectives as theirs (i.e., execution *time* for *Time Difference* (*TD*) and fault history for *Mean Probability* (*MPO*)). However, we have defined other objectives (i.e., *Mean Priority* (*MPR*) and *Mean Consequence* (*MC*)) along with the time budget, and user preferences for objectives different to their work.

Search algorithms have been extensively applied for other testing problems as well [29-31]. For instance, 1) Test suite minimization where three weight-based search algorithms were evaluated by defining three effectiveness measures [32]; 2) Test case generation by taking three objectives into account [33]; 3) Test case prioritization where the authors [34] defined one cost measure and three effectiveness measures. Our work is different due to different motivation, i.e., we focus on test case selection. Furthermore, totally different cost/effectiveness measures were defined in our work as compared with the above-mentioned existing works.

8 Conclusion

This paper proposed a search-based test case selection approach by defining one cost measure, i.e., *Time Difference* (*TD*) and three effectiveness measures, i.e., *Mean Priority* (*MPR*), *Mean Probability* (*MPO*) and *Mean Consequence* (*MC*) based on the real requirements from our industrial partner. We empirically evaluated eight existing multi-objective search algorithms, with two weight assignment strategies (e.g., Fixed Weights). The results show that Strength Pareto Evolutionary Algorithm 2 (SPEA2) significantly outperforms the other search algorithms, and it managed to improve on average 32.7%, 39% and 33% for *MPR*, *MPO* and *MC* respectively as compared to random search. In the near future, we plan to apply other evaluation metrics (e.g., epsilon [35]) to evaluate the Pareto-based algorithms in a more thorough way. We also want to hybridize evolutionary algorithms with local search algorithms to see whether a better performance can be achieved for solving our test case selection problem.

Acknowledgement

This research was supported by the Research Council of Norway (RCN) funded Certus SFI. Shuai Wang is also supported by RFF Hovedstaden funded MBE-CR project. Tao Yue and Shaukat Ali are also supported by RCN funded Zen-Configurator project, the EU Horizon 2020 project funded U-Test, RFF Hovedstaden funded MBE-CR project and RCN funded MBT4CPS project.

References

[1] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering,* vol. 28, no. 2, pp. 159-182. 2002.

[2] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," *Department of Computer Science, King's College London, Tech. Rep. TR-09-03,* 2009.

[3] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety,* vol. 91, no. 9, pp. 992-1007. Elsevier, 2006.

[4] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering,* vol. 16, no. 8, pp. 870-879. 1990.

[5] S. Droste, T. Jansen, and I. Wegener, "On the analysis of the (1+ 1) evolutionary algorithm," *Theoretical Computer Science,* vol. 276, no. 1, pp. 51-81. 2002.

[6] C. A. C. Coello, D. A. Van Veldhuizen, and G. B. Lamont, *Evolutionary algorithms for solving multi-objective problems* vol. 242: Springer, 2002.

[7] K. Deb, *Multi-objective optimization using evolutionary algorithms* vol. 16: John Wiley & Sons, 2001.

[8] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* vol. 6, no. 2, pp. 182-197. IEEE, 2002.

[9] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "Design issues in a multiobjective cellular genetic algorithm," in *Evolutionary multi-criterion optimization*, pp. 126-140. Springer, 2007.

[10] E. Zitzler, M. Laumanns, and L. Thiele, "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), 2001.

[11] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba, "Solving three-objective optimization problems using a new hybrid cellular genetic algorithm," in *Parallel Problem Solving from Nature–PPSN X*, ed: Springer, 2008, pp. 661-670.

[12] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *Parallel Problem Solving from Nature-PPSN VIII*, pp. 832-842. Springer, 2004.

[13] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach," *IEEE Transactions on Evolutionary Computation,* vol. 3, no. 4, pp. 257-271. IEEE, 1999.

[14] S. Ali and T. Yue, "Evaluating Normalization Functions with Search Algorithms for Solving OCL Constraints," in *Testing Software and Systems*, ed: Springer, 2014, pp. 17-31.

[15] *OREDA Offshore Reliability Data Handbook 2002, 4th edition* Høvik, Norway : OREDA Participants : Distributed by Der Norske Veritas, 2002.

[16] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "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*, pp. 631-642. ACM, 2016.

[17] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132. SAGE journals, 2000.

[18] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics,* pp. 50-60. 1947.

[19] A. Arcuri and L. Briand, "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)*, pp. 1-10. IEEE, 2011.

[20] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software,* vol. 42, no. 10, pp. 760-771. Elsevier, 2011.

[21] *Results of the experiments available publicly in website*. Available: http://zen-tools.com/conference/GECCO2016.html

[22] *The Abel computer cluster*. Available: http://www.uio.no/english/services/it/research/hpc/abel/

[23] M. de Oliveira Barros and A. Neto, "Threats to Validity in Search-based Software Engineering Empirical Studies," *UNIRIO-Universidade Federal do Estado do Rio de Janeiro, techreport,* vol. 6, 2011.

[24] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Proceedings of the 3rd International Symposium on Search Based Software Engineering*, pp. 33-47. Springer, 2011.

[25] I. Statistical Sciences, *S-PLUS Guide to Statistical and Mathematical Analysis, Version 3.3*: StatSci, a division of MathSoft, Incorporated, 1995.

[26] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pp. 119-129. IEEE, 2002.

[27] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 international symposium on Software testing and analysis*, pp. 1-12. ACM, 2006.

[28] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 140-150. ACM, 2007.

[29] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software,* vol. 103, pp. 370-391. 2015.

[30] S. Wang, S. Ali, A. Gotlieb, and M. Liaaen, "A systematic test case selection methodology for product lines: results and insights from an industrial case study," *Empirical Software Engineering,* pp. 1-37. 2014.

[31] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-Objective

Search," presented at the International Conference on Software Engineering (ICSE), 2016.

[32] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pp. 1493-1500. ACM, 2013.

[33] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Multi-objective test generation for software product lines," in *Proceedings of the 17th International Software Product Line Conference*, pp. 62-71. ACM, 2013.

[34] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen, "Multi-objective test prioritization in software product line testing: an industrial case study," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pp. 32-41. ACM, 2014.

[35] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. Da Fonseca, "Performance assessment of multiobjective optimizers: an analysis and review," *Evolutionary Computation, IEEE Transactions on,* vol. 7, no. 2, pp. 117-132. 2003.

Paper C

CBGA-ES^{+}: A Cluster-Based Genetic Algorithm With Non-Dominated Elitist Selection for Supporting Multi-objective Test Optimization

Dipesh Pradhan, Shuai Wang, Shaukat Ali,
Tao Yue, Marius Liaaen

Published in IEEE Transactions on
Software Engineering (TSE), November 2018.
DOI: 10.1109/TSE.2018.2882176

The layout has been revised.

Abstract

Many real real-world test optimization problems (e.g., test case prioritization) are multi-objective intrinsically and can be tackled using various multi-objective search algorithms (e.g., Non-dominated Sorting Genetic Algorithm (NSGA-II)). However, existing multi-objective search algorithms have certain randomness when selecting parent solutions for producing offspring solutions. In a worst case, suboptimal parent solutions may result in offspring solutions with bad quality, and thus affect the overall quality of the solutions in the next generation. To address such a challenge, we propose CBGA-ES$^+$, a novel cluster-based genetic algorithm with non-dominated elitist selection to reduce the randomness when selecting the parent solutions to support multi-objective test optimization. We empirically compared CBGA-ES$^+$ with random search and greedy (as baselines), four commonly used multi-objective search algorithms (i.e., Multi-objective Cellular genetic algorithm (MOCell), NSGA-II, Pareto Archived Evolution Strategy (PAES), and Strength Pareto Evolutionary Algorithm (SPEA2)), and the predecessor of CBGA-ES+ (named CBGA-ES) using five multi-objective test optimization problems with eight subjects (two industrial, one real world, and five open source). The results showed that CBGA-ES+ managed to significantly outperform the selected search algorithms for a majority of the experiments. Moreover, for the solutions in the same search space, CBGA-ES$^+$ managed to perform better than CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2 for 2.2%, 13.6%, 14.5%, 17.4%, and 9.9%, respectively. Regarding the running time of the algorithm, CBGA-ES+ was faster than CBGA-ES for all the experiments.

Keywords: Elitist Selection; Multi-Objective Genetic Algorithm; Multi-Objective Test Optimization; Search.

1 Introduction

Many real-world test optimization problems are multi-objective intrinsically, which requires considering multiple conflicting objectives when finding optimal solutions. For instance, based on our long-term collaboration with Cisco Systems [1, 2], we identified a test case prioritization problem [3] with four conflicting objectives (e.g., fault detection capability) to prioritize a given number of test cases into an optimal order. Another example is the testing resource allocation problem [4, 5] (i.e., allocating test resources optimally to different software modules), to minimize testing cost (e.g., testing time) and maximize the reliability of the modules.

Search-Based Software Testing (SBST) has been widely applied to address various multi-objective test optimization problems [6-9]. The foundation of employing SBST is to formulate a testing problem into a mathematical optimization problem, which can be efficiently solved with metaheuristic optimization algorithms (e.g., Genetic Algorithm) [10-12]. Existing studies [8, 12-14] have shown that multi-objective search algorithms

(e.g., Non-dominated Sorting Genetic Algorithm II (NSGA-II) [15]) are effective in solving different multi-objective test optimization problems, such as test suite minimization and test case prioritization..

However, based on our experience of applying SBST techniques for addressing several multi-objective test optimization problems [2, 3, 16, 17], we observed that the representative set of multi-objective search algorithms make choices based on random number generation when selecting parent solutions (i.e., stochastic parent selection) to produce offspring solutions due to selection mechanisms employed in the algorithms [18]. For example, in binary tournament selection (commonly used in the literature [19, 20]), two solutions are randomly selected, and the better solution is selected as the parent. However, if the selected parent solutions are suboptimal in the population, it might result in offspring solutions with bad quality (i.e., the produced offspring solutions have worse values for the different objectives as compared to the solutions in the population). This may subsequently degrade the overall quality of the solutions in the next generation, and in the worst case, stochastic parent selection may prevent algorithms in finding optimal solutions.

We argue that an efficient elitist strategy can largely reduce such randomness when selecting parent solutions to produce offspring solutions. Thus, we propose a Cluster-Based Genetic Algorithm with Non-Dominated Elitist Selection (CBGA-ES$^+$) for supporting multi-objective test optimization. The core idea of CBGA-ES$^+$ lies in 1) dividing the initial population into different clusters, 2) defining *cluster dominance strategy* to rank the different clusters, and 3) selecting the non-dominated solutions from the best clusters for producing offspring solutions. When a new population is created, this process will be repeated for producing the next generation until the termination conditions for the algorithm are met. CBGA-ES$^+$ extends our previous algorithm, Cluster-Based Genetic Algorithm with Elitist Selection (CBGA-ES) proposed in our conference paper [21]. The core difference between CBGA-ES$^+$ and CBGA-ES lies in the strategy of selecting elite solutions, i.e., CBGA-ES$^+$ selects only the non-dominated solutions from the best clusters while CBGA-ES selects all the solutions from the best clusters. Such a selection strategy lets CBGA-ES$^+$ select better elite solutions from the population and thus improve the quality of offspring solutions. More specifically, two sub-algorithms are defined in CBGA-ES$^+$ for selecting the non-dominated elite solutions (Section 3.2).

We empirically evaluated CBGA-ES$^+$ with five multi-objective test optimization problems (i.e., test suite minimization, test case prioritization, test case selection, testing resource allocation, and integration and test order problem) by employing eight subjects (two industrial, one real-world, and five subjects from open source projects) for a total of 20 experiments. A comprehensive empirical evaluation was performed to compare CBGA-ES$^+$ with 1) baseline algorithms, i.e., random search (RS) and Greedy; 2) four representative multi-objective search algorithms from the literature [2, 12]: (NSGA-II) [15], Strength Pareto Evolutionary Algorithm (SPEA2) [22], Pareto Archived Evolution Strategy (PAES) [23], and Multi-objective Cellular Genetic Algorithm (MOCell) [24]; and

3) its predecessor, CBGA-ES [21]. These algorithms were selected because of their success in previous literature [2, 3, 5, 17, 25].

The results showed that CBGA-ES$^+$ managed to significantly outperform the baseline algorithms (RS and Greedy), the four selected multi-objective search algorithms (e.g., NSGA-II) and CBGA-ES for the majority of experiments. Specifically, CBGA-ES$^+$ significantly outperformed 1) RS for 100%, 2) Greedy for 85%, and 3) the five selected search algorithms for an average of 66% of the experiments. Overall, CBGA-ES$^+$ managed to perform better than CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2 for 2.2%, 13.6%, 14.5%, 17.4%, and 9.9%, respectively for solutions within the same search space for the five test optimization problems. Moreover, the solutions produced by CBGA-ES$^+$ had a better fault detection rate as compared to the selected algorithms for the majority of the experiments. Additionally, the running time of CBGA-ES$^+$ is faster than CBGA-ES for all the experiments.

This paper extends our previous work [21] with the following key improvements:

1) CBGA-ES$^+$ is extended from its predecessor (CBGA-ES [21]) by introducing non-dominated elitist selection strategy, i.e., selecting only the non-dominated elite solutions to form the elite population when producing offspring solutions (Section 3.2);
2) The empirical evaluation has been extended by involving: 1) two additional multi-objective test optimization problems (i.e., test resource allocation, and integration and test order) with two open source subjects; and 2) three additional open source subjects for test suite minimization problem, test case prioritization problem, and test case selection problem (Section 4);
3) Four additional metrics have been added to evaluate the quality of the algorithms (Section 6);
4) A more in-depth discussion has been added based on the results (Section 7);
5) Background and related work sections have been enhanced by a) introducing the five multi-objective test optimization problems in detail, b) discussing the working mechanisms of different multi-objective search algorithms (e.g., NSGA-II), and c) including more comparisons between CBGA-ES$^+$ and the existing work (Section 2 and Section 9).

The remainder of the paper is organized as follows. Section 2 gives relevant background, and Section 3 describes our proposed algorithm, CBGA-ES$^+$ for supporting multi-objective test optimization. The eight subjects are described in Section 4, and Section 5 describes the empirical study design. Section 6 presents the results of the empirical study, and overall discussion is presented in Section 7. Section 8 presents the threats to validity. Related work is reported in Section 9, and we conclude this paper in Section 10.

2 Background

2.1 Multi-Objective Test Optimization

In Search-Based Software Testing (SBST), multi-objective test optimization aims to find tradeoff solutions among multiple conflicting objectives (e.g., execution cost, effectiveness) for various software testing problems (e.g., test suite minimization (*TSM*) [2, 11]). Since there might exist more than one best solution, a set of solutions with equivalent quality (i.e., non-dominated solutions) is usually produced based on *Pareto optimality*, which is called as Pareto fronts [26-28]. More specifically, *Pareto optimality* defines the Pareto dominance to assess the quality of solutions [29]. Suppose there are m objectives $O = \{o_1, o_2, \dots, o_m\}$ to be optimized for a multi-objective test optimization problem (e.g., test case prioritization (*TCP*)) and each objective can be measured using a fitness function f_j from $F = \{f_1, f_2, \dots, f_m\}$. If we aim to minimize the fitness function such that a lower value for an objective implies better performance, then solution A dominates solution B (i.e., $A \succ B$) *iff* $\forall_{i=1,2,\dots,n} f_i(A) \leq f_i(B) \wedge \exists_{i=1,2,\dots,n} f_i(A) < f_i(B)$.

Fig. C-1 presents a graphical representation of Pareto dominance for an optimization problem with two objectives for a minimization problem (i.e., a lower objective value is better) with five solutions: *A*, *B*, *C*, *D*, and *E*. In Fig. C-1, *A* dominates *B*, *C*, and *D* since the values of both the objectives (i.e., min F_1 and min F_2) for *A* is lower than *B*, *C*, and *D*. However, *E* and *A* are non-dominated solutions since for the objective function min F_1, *E* is better than *A* while *A* is better than *E* for the objective function min F_2.

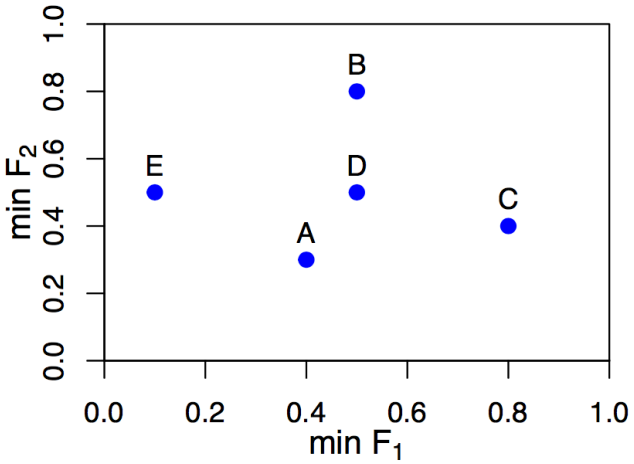

Fig. C-1. Pareto dominance for a minimization problem with two objectives

2.2 Genetic Algorithms

Metaheuristics combine basic heuristic methods in higher level frameworks to find solutions to combinatorial problems at a reasonable computational cost [8, 30]. Genetic Algorithms (GAs) are a form of metaheuristic inspired by the process of natural selection that optimize one or more objectives (e.g., maximizing code coverage while minimizing

the execution cost of test cases) using one or more fitness functions to assess the quality of solutions. GAs employ bio-inspired operators (i.e., *selection*, *crossover*, and *mutation*) for producing offspring solutions [31]. The *selection* operator selects best solutions based on the fitness function, while the *crossover* operator partially exchanges the selected two parent solutions. The *mutation* operator mutates a given solution by randomly changing parts (e.g., a test case) of the solution.

We chose four representative multi-objective search algorithms from the literature [2, 12]: Non-dominated Sorting Genetic algorithm II (NSGA-II) [15], Strength Pareto Evolutionary Algorithm (SPEA2) [22], Pareto Archived Evolution Strategy (PAES) [23], and Multi-objective Cellular Genetic Algorithm (MOCell) [24] for the evaluation. The selected four search algorithms are designed based on *Pareto optimality* (Section 2.1), and they have achieved good results for addressing multi-objective test optimization problems in state-of-the-art [2, 3, 5, 17, 25] as shown in Table C-1. The column *Applied To* in Table C-1 shows the multi-objective test optimization problem (out of the five considered multi-objective test optimization problems) where the listed algorithm has been applied in the literature and achieved good results.

Table C-1. Classification of the Selected Search Algorithms*

Algorithm Category		Algorithm	Applied To
GA	Sorting-Based	NSGA-II	*TSM, TCP, TCS, TRA, ITO*
	Cellular-Based	MOCELL	*TSM, TCP, TCS, TRA*
Strength Pareto EA		SPEA2	*TSM, TCP, TCS, ITO*
Evolution Strategies		PAES	*TSM, TCP, TCS, TRA, ITO*

**TSM*: Test Suite Minimization, *TCP*: Test Case Prioritization, *TCS*: Test Case Selection, *TRA*: Testing Resource Allocation, *ITO*: Integration and Test Order.

In NSGA-II the solutions (chromosomes) in the population are sorted and placed into several fronts based on the ordering of Pareto dominance [15]. The individual solutions are selected from the non-dominated fronts, and if the number of the 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 with the others in the population [32].

In SPEA2 the fitness value for each solution is calculated by adding 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) [22]. Initially, SPEA2 creates an empty archive that is then filled by the non-dominated solutions from the population, and in the subsequent generations, the solutions from the archive and the non-dominated solution in the current population create a new population. If the combined non-dominated solutions are 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.

MOCell is based on the cellular model of genetic algorithms with an assumption that an individual solution can only interact with its neighbors during the search process (neighborhood) [24]. More specifically, MOCell stores a set of non-dominated solutions in an external archive, and after each generation, MOCell replaces a fixed number of

randomly chosen individuals of the populations by solutions from the archive with a feedback procedure. Note that the replacement only occurs if the solutions from the population are worse than the solutions in the archive.

PAES maintains an archive of non-dominated solutions. Initially, a random solution is added to the archive, and it is used to generate the offspring solution [23]. The newly generated solution is used to replace the parent solution if it is better, and it is further added to the archive if it is better than the existing solutions. If the generated solution is worse than the parent solution, it is discarded, and the parent solution is used to generate another solution. However, if the generated solution is neither dominating nor dominated by the archive, additional measures (similar to NSGA-II) are taken into account: if the generated solution lies in a more crowded part of the feasible place (with respect to members of the archive) as compared to the parent solution, it is disposed, else it replaces the parent solution.

2.3 Greedy Algorithm

The greedy algorithm works on the "next best" search principle, such that the element with the highest weight (e.g., statement coverage) is selected first [8]. It is then followed by the element with the second highest weight and so on until all the elements are selected, or termination criteria of the algorithm are met (e.g., the total execution time of the selected test cases meets the time budget). If there exist multiple elements with the same weight, one of them is randomly selected [33]. The greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized (maximized or minimized). Note that Greedy algorithm does not go back and reverse the direction.

For multi-objective optimization, Greedy algorithm converts a multi-objective optimization problem into a single optimization problem using the weighted-sum method [34] for fitness assignment, such that each objective is given equal weight (if all the objective holds equal importance). After that, the weight of each element is obtained by summing up the weighted objective values.

2.4 Five Multi-Objective Test Optimization Problems

We investigated in total five multi-objective test optimization problems: test suite minimization (*TSM*), test case prioritization (*TCP*), test case selection (*TCS*), testing resource allocation (*TRA*), and integration and test order (*ITO*) problem. All five multi-objective test optimization problems have been widely investigated by the state-of-the-art [25, 35-38]. Notice that *TSM*, *TCP*, and *TCS* problems were extracted from two different domains (*telecommunication* and *maritime*) based on our collaboration with industrial partners, while *TRA* and *ITO* problems were obtained from a well-known Search-based Software Engineering Repository hosted by the CREST center [39]. We formally present each multi-objective test optimization problem in detail as below, and the details of the objective functions are provided in Section 4.

2.4.1 Test Suite Minimization (TSM) Problem

Definition 1. *Multi-Objective Test Suite Minimization:* For a test suite $T = \{t_1, t_2, \dots, t_n\}$ and a vector of m objective functions $F = \{f_1, f_2, \dots, f_m\}$, the problem consists in finding a *Pareto optimal* set of minimized test suites $T' \subset T$.

The objective functions can be defined as various minimization criteria (e.g., higher code coverage [11]).

2.4.2 Test Case Prioritization (TCP) Problem

Definition 2. *Multi-Objective Test Case Prioritization*. Given a test suite $T = \{t_1, t_2, \dots, t_n\}$, the set of permutations of T: PT, and a vector of m objective functions: $F = \{f_1, f_2, \dots, f_m\}$, the problem consists in finding a *Pareto optimal* set of prioritized test suites $T' \subset T$ such that $T' \in PT$ [14].

The objective functions represent the concerned prioritization criteria (e.g., higher statement coverage [10]). The test cases need to be executed in the order of the permutation that *TCP* produces, but testing can be terminated at any point in the testing process.

2.4.3 Test Case Selection (TCS) Problem

Definition 3. *Multi-Objective Test Case Selection:* Given a test suite $T = \{t_1, t_2, \dots, t_n\}$, and a vector of m objective functions, $F = \{f_1, f_2, \dots, f_m\}$, the problem consists in finding a *Pareto optimal* set of test suites with selected test cases $T' \subset T$.

The objective functions can be defined as various selection criteria (e.g., execution cost [12]). Both *TSM* and *TCS* aim at choosing a subset of test cases from the existing test suite at the same time achieving pre-defined objectives. However, *TSM* eliminates the redundant test cases from the existing test suite for the current systems to reduce the cost of testing (e.g., cost), while *TCS* focuses on selecting a subset of test cases from the existing test suite to test a modified version of the systems [2, 37]. Therefore, we have classified *TSM* and *TCS* as separate problems as is often done in literature [37].

2.4.4 Testing Resource Allocation (TRA) Problem

Definition 4. *Multi-Objective Testing Resource Allocation:* Given a set of o modules, $M = \{m_1, m_2, \dots, m_o\}$, and a vector of m objective functions: $F = \{f_1, f_2, \dots, f_m\}$, let $s_a = \{h_{a1}, h_{a2}, \dots, h_{ao}\}$ be a solution such that h_{ao} represents a set of testing hours to be allocated to module m_o, and $\sum_{i=the\ 1}^{o} h_{ai} \leq H_{max}$, where H_{max} is the maximum testing hours available. The *TRA* problem finds a *Pareto optimal* set of solutions with respect to the objective functions, $F = \{f_1, f_2, \dots, f_m\}$.

TRA aims to optimally allocate the resources to different modules (that comprise the software system) for maximizing the reliability while minimizing the testing resources (e.g., cost, testing effort) [5]. *TRA* problem is not a permutation of the solution, but rather an allocation of different testing hours to the modules such that the total allocated hours for

the modules is less or equal to the maximum allocated hours and modules with higher reliability are allocated more testing hours.

2.4.5 Integration and Test Order (ITO) Problem

Definition 5. *Multi-Objective Integration and Test Order:* Given a set of units (i.e., classes or aspects), $U = \{u_1, u_2, \dots, u_p\}$, the set of permutations of U: PU, and a vector of m objective functions: $F = \{f_1, f_2, \dots, f_m\}$. The *ITO* aims to find a *Pareto optimal* set of solutions with respect to the objective functions, $F = \{f_1, f_2, \dots, f_m\}$, such that each solution is an element of PU.

The objective function can be defined as various criteria (e.g., the number of attributes that need to be emulated if the dependency between the classes is broken [40]). *ITO* problem focuses on determining an order/sequence to integrate and test the units (e.g., classes) to minimize the stubbing cost, where a stub is an emulation of a unit that has not yet been implemented or integrated into the software [25]. By rearranging the ordering of the units, the most required units can be integrated and tested first so that the next dependent units do not require a stub for that unit [35]. Even though both *TCP* and *ITO* problem include a permutation of the elements, they are different. This is because in *TCP*, irrespective of the test case order, test case time does not change, while, the coverage is dependent on the test case order. However, in *ITO* there are dependencies among certain units for all the objectives, and *ITO* aims to test the most required units first so that it is not required to create a stub for the dependent unit. Therefore, in *ITO* if the required units are already tested (i.e., they are scheduled earlier), the later units have no cost to develop stub. Moreover, in *ITO* if two units are independent of one another, their order does not matter.

3 CBGA-ES$^+$ Algorithm

The core idea of CBGA-ES$^+$ is to cluster a given number of candidate solutions (i.e., population) and select non-dominated elite ones for producing offspring solutions. This is done by grouping solutions with similar fitness into a cluster, sorting the clusters based on the *cluster dominance strategy* (defined in Section 3.1), and selecting the non-dominated solutions from the best clusters for the next generation for reproduction. We first introduce our *cluster dominance strategy* in Section 3.1 followed by the description of CBGA-ES$^+$ in Section 3.2.

3.1 Cluster Dominance Strategy

The *cluster dominance strategy* is defined to identify the dominance relationship between two clusters in terms of pre-defined objectives for a multi-objective problem (MOP). Each cluster is composed of a set of similar candidate solutions for solving the MOP and holds a center that is a vector containing a specific value for each objective. The value for each center is the mean fitness of all the included solutions in the cluster for a particular

objective [41]. We illustrate our cluster dominance strategy with minimization problems, i.e., a solution with a lower value for an objective implies better performance than a solution with a higher value.

Suppose there are two clusters c_a and c_b with centers $m_a = \{m_{a1}, m_{a2}, \dots, m_{an}\}$ and $m_b = \{m_{b1}, m_{b2}, \dots, m_{bn}\}$, where n is the number of objectives to optimize, we define *cluster dominance* and *cluster partial dominance* below.

Cluster Dominance. c_a dominates c_b (i.e., $c_a \succ c_b$) *iff*

$$\forall_{i=1,2,\dots,n} m_{ai} \leq m_{bi} \wedge \exists_{i=1,2,\dots,n} m_{ai} < m_{bi}$$

m_{ai} and m_{bi} refer to the values of the i^{th} objective in m_a and m_b, respectively.

Cluster Partial Dominance. c_a partially dominates c_b (i.e., $c_a \succcurlyeq c_b$) *iff* one of the following two cases hold *true*.

Case1. $n\,|\{i = 1,2,\dots,n | m_{ai} < m_{bi}\}| > |\{i = 1,2,\dots,n | m_{ai} > m_{bi}\}|$, where $|\{i = 1,2,\dots,n | m_{ai} < m_{bi}\}|$ is the number of objectives in which the values of m_a are lower (better) than values in m_b, and $|\{i = 1,2,\dots,n | m_{ai} > m_{bi}\}|$implies the number of objectives in which the values of m_b are better than values in m_a.

Case2. If $|\{i = 1,2,\dots,n | m_{ai} < m_{bi}\}| = |\{i = 1,2,\dots,n | m_{ai} > m_{bi}\}|$,

1) $\left(\sum_{i=1}^{n} \frac{m_{bi} - m_{ai}}{m_{bi}}\right) > 0$ if $\forall_{i=1,2,\dots,n} m_{bi} > 0$

2) $\left(\sum_{i=1}^{n} \frac{m_{ai} - m_{bi}}{m_{ai}}\right) < 0$ if $\forall_{i=1,2,\dots,n} m_{ai} > 0 \wedge \exists_{i=1,2,\dots,n} m_{bi} = 0$

3) $(\sum_{i=1}^{n} m_{bi} - m_{ai}) > 0$ if $\exists_{i=1,2,\dots,n} m_{bi} = 0 \wedge \exists_{i=1,2,\dots,n} m_{ai} = 0$

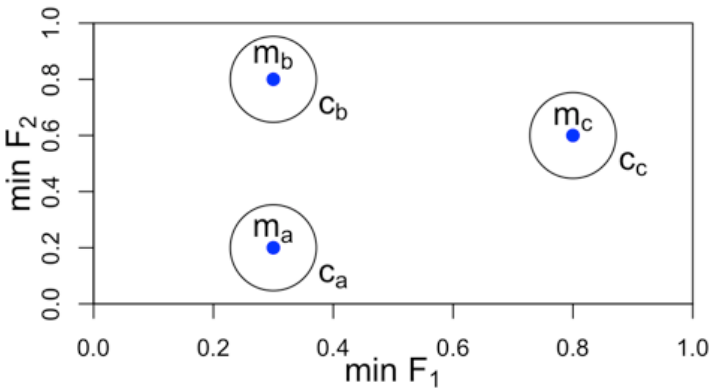

Fig. C-2. Three clusters for a two objective minimization problem

For instance, Fig. C-2 represents a two objective minimization problem consisting of three clusters: c_a, c_b, and c_c with centers m_a, m_b, and m_c, such that $m_a = \{0.3, 0.2\}$, $m_b = \{0.3, 0.8\}$, and $m_c = \{0.8, 0.6\}$. Based on our cluster dominance strategy, c_a dominates both c_b and c_c since all the values for m_a are equal or lower than the values for m_b and m_c. Similarly, based on our cluster partial dominance strategy, c_b dominates c_c since both m_b and m_c have better values for one objective (e.g., $0.3 < 0.8$ and $0.8 > 0.6$), however, the difference of two objectives is positive for m_c, i.e., case 2, scenario 1.

3.2 CBGA-ES$^+$ Algorithm

Recall that CBGA-ES$^+$ is designed to address various multi-objective test optimization problems. Thus, the input for CBGA-ES$^+$ includes an original test suite T to be optimized and a set of parameters to be configured, i.e., population size, the number of clusters, and elite population minimum size. Elite population minimum size was defined to prevent premature convergence of the algorithm, which might arise due to the introduced elitist selection, as discussed in [42]. More specifically, if the number of elite solutions is small (e.g., two), it might prevent the diversity of the solutions in the population, and thus the algorithm might not be able to produce optimal solutions. We present the pseudo code of CBGA-ES$^+$ in Algorithm C-1.

CBGA-ES$^+$ starts by creating an initial set of a random population (P_t) of the size I (line 2 of Algorithm C-1). Afterward, *Lloyd's algorithm* (which is a commonly-used k-means clustering algorithm [41, 43]) is used to cluster solutions with similar fitness values for n objectives into K clusters (line 4). More specifically, Lloyd's algorithm as shown in Algorithm C-2, first randomly chooses one solution from P_t for each cluster in C and treats the objective values of the solution (n objectives) as the cluster centers, respectively. Notice that the selected solutions for each cluster should be different such that no two clusters have the same centers. Furthermore, Euclidean distance is used to measure the distance between the cluster centers and a particular solution s_j from P_t (line 7), and the solution s_j is added to a cluster with the least Euclidean distance between s_j and the center of the cluster (lines 8-9).

Algorithm C-1: CBGA-ES$^+$

Input: Original test suite *T*, population size I, number of clusters K, elite population minimum size E
Output: Optimal test suite solutions S
Begin

```
1   t ⟵ 0                                              // current generation
2   P_t ⟵ Random-Population (I)
3   while not (termination_conditions_satisfied) do
4       C ⟵ cluster P_t into K clusters using Lloyd's algorithm
5       C' ⟵Sort-Clusters (C)                          // sort the clusters using the cluster
                                                          dominance strategy
6       P_e ⟵ ∅                                        // initialize the elite population
7       P_e ⟵ Update-elite-population (P_e, c_1', |c_1'|)
8       while (|P_e| < E) do
9           k ⟵ k + 1
10          Update-elite-population (P_e, c_k', E)
11      Q_t ⟵ Apply crossover and mutation operators to P_e
12      P_{t+1} ⟵ P_e ∪ Q_t
13      t ⟵ t + 1                                      // increase the generation
14  S ⟵ P_e
```

Once all the solutions are partitioned into k clusters (lines 6-9 in Algorithm C-2), each cluster center m_i is updated by taking the mean value of all the included solutions in the cluster for each objective (line 10 in Algorithm C-2). When the values of cluster centers change, all the solutions are re-partitioned into k clusters by measuring the Euclidean distance between the solutions and the new cluster centers (lines 6-9 in Algorithm C-2). Afterward, all the cluster centers will be updated again, and such process is repeated until the values in the cluster center do not change in two consecutive iterations (lines 11-12 in Algorithm C-2). Finally, the produced clusters are returned (line 11), such that each cluster consists of a set of similar solutions with respect to the pre-defined objectives.

Algorithm C-2: Lloyd's algorithm [41]

Input: Population $P_t = \{s_1, \dots, s_l\}$, number of clusters K
Output: Solutions partitioned into clusters C
Begin
1 Randomly pick solutions from P_t as the centers for C
2 $C = \{c_1, \dots, c_k\}$ // K clusters
3 $m_i \longleftarrow$ center for cluster c_i
4 **while** (true) **do**
5 $oldC = m$ // store old center values in a variable
6 **for** ($s_j \in P_t$) **do**
7 $\forall_{i=1,2,\dots,k} d_{s_j m_i} = \left\| s_j - m_i \right\|_2$
8 $i \longleftarrow argmin_{1,2,\dots,k} d_{s_j m_i}$
9 $c_i \longleftarrow c_i \cup s_j$ // add solution to the cluster
10 $\forall_{i=1,2,\dots,k}\, m_i = \frac{1}{n_i} \sum_{s_a \in C_i} s_a$ // recalculate cluster center as the mean of the solutions in the cluster
11 **if** $oldC = m$ **then** // check if cluster center changed
12 break
13 return C

Algorithm C-3: Update-elite-population

Input: Elite population P_e, cluster c, population size s
Output: Elite population P_e with solutions added from c
Begin
1 **for** ($s_i \in c$) **do**
2 $add \longleftarrow true$
3 **for** ($s_j \in c$ & $i \neq$ j) **do**
4 $result \longleftarrow$ Dominance-comparator (s_i, s_j)
5 **if** $result = true$ **then**
6 $add \longleftarrow false$
7 **break**
8 **if** $add = true$ **then**
9 $P_e \longleftarrow P_e \cup s_i$
10 **if** ($|P_e| = s$) **then**
11 **break**
12 return P_e

Afterward, the clusters obtained from Lloyd's algorithm are sorted based on the *cluster dominance strategy* (i.e., *cluster dominance* and *cluster partial dominance*) defined in Section 3.1, and the non-dominated solutions are added to the elite population from the cluster that dominates the other (line 7 in Algorithm C-1) using the algorithm *update-elite –population* (Algorithm C-3) that in turn uses the algorithm *dominance-comparator* (Algorithm C-4). More specifically, the algorithm *update-elite-population* adds the non-dominated solutions from the best clusters to the elite population by comparing each solution with all other solutions in the cluster (line 4 in Algorithm C-3) using the algorithm *dominance-comparator* (Algorithm C-4).

The algorithm *dominance-comparator* compares the values for each objective between two solutions s_a and s_b (lines 4-8 in Algorithm C-4) to check if the solution s_a is dominated by the solution s_b followed by returning the result of the comparison (line 9 in Algorithm C-4). If the solution is not dominated by any solutions in the cluster, it is added to the elite population (line 9 in Algorithm C-3). This process is repeated until all the solutions in the cluster are compared with one other or the size of the elite population is equal to the specified population size. After this, the elite population is returned in line 12 in Algorithm C-3. If the size of the returned elite population is smaller than the specified minimum elite population size E, solutions from the next dominant cluster are chosen for the elite population using the same algorithm *update-elite-population* (Algorithm C-3) until the size of the elite population is equal to E (lines 8 - 10 in Algorithm C-1).

Algorithm C-4: Dominance-comparator

Input: Two solutions to compare s_a and s_b, $O = \{o_1, \dots, o_n\}$ the set of n objectives to optimize
Output: Boolean result of comparison
Begin

```
1   dominated_a ⟵ false
2   dominated_b ⟵ false
3   for (o_i ∈ O) do
4      if s_ai < s_bi then
5         dominated_b ⟵ true
6      else if s_ai > s_bi then
7         dominated_a ⟵ true
8   return dominated_a && ! dominated_b
```

Once the size of the elite population P_e is fulfilled (i.e., the size E is reached), *crossover* and *mutation* are applied to the elite population for generating offspring solutions (line 11 in Algorithm C-1). More specifically, with respect to *crossover*, two parent solutions are selected at random from P_e to produce offspring Q_t by swapping parts (e.g., test cases) from both of the parents. Moreover, the *mutation* operator is applied to randomly mutate test cases of a specific solution to obtain the offspring solution. Subsequently, Q_t and P_e are combined to form the population P_{t+1} for the next generation without involving any replacement operator (line 12 in Algorithm C-1). This process of clustering the new population P_{t+1}, selecting the elite population, and generating offspring solutions is

repeated until termination conditions for the algorithm are met, e.g., time budget for the algorithm. Once the termination conditions are satisfied, the elite solutions are given as optimal solutions for the selected test optimization problems (line 15 in Algorithm C-1).

CBGA-ES$^+$ is an extended algorithm based on the algorithm proposed in our previous work named CBGA-ES [21]. The key improvement of CBGA-ES$^+$ compared with CBGA-ES is that CBGA-ES$^+$ has a non-dominated elitist selection strategy (Algorithms 3 and 4), which borrows core concepts from the literature [15, 22, 24]. This selection strategy helps CBGA-ES$^+$ to select only non-dominated solutions from the best cluster, followed by the second best cluster and so on, while CBGA-ES selects all the solutions from the best cluster, followed by the second best and so on. In case the number of solution in the cluster is higher than the specified elite population size, random solutions are selected [21], which reduces the probability of non-dominated solutions being selected. Thus, CBGA-ES$^+$ can choose better elite solutions than CBGA-ES and improve the quality of offspring solutions. For instance, suppose the cluster consists of 20 solutions in total with only one non-dominated solution. If only one more solution can be selected based on the specified elite population size, the probability of including the non-dominated solution into the elite population is only 5% for CBGA-ES, while CBGA-ES$^+$ can have 100% probability of including this non-dominated solution into the elite population.

4 Subjects

To evaluate CBGA-ES$^+$, we performed a total of 20 experiments with eight subjects for the five multi-objective test optimization problems (e.g., *TSM* defined in Section 2.4). The subjects include 1) Two industrial subjects [2, 3] from the telecommunication domain; 2) One real-world subjects study from the maritime domain [17]; and 3) Five subjects from open source project from [4, 5, 25, 35, 44]. The details of the subjects are summarized in Table C-2.

4.1 Industrial Subjects (D_1 and D_2)

The industrial subjects belong to the telecommunication domain. Our industrial partner Cisco Systems, Norway develops a product line of Video Conferencing Systems (VCSs) [2]. There is an average of three million lines of embedded C code in each VCS [16], and each VCS requires thorough testing before releasing them in the market. However, testing is expensive at Cisco (e.g., executing test cases need to set up different hardware and network environment), and thus, it requires optimizing the testing process to reduce the cost of testing while preserving effectiveness.

1) D_1: D_1 consists of 489 test cases and based on our industrial collaboration in previous work for *TSM* [2], we derived five objectives as shown in Table C-2: maximizing test minimization percentage (*TMP*), feature pairwise coverage (*FPC*), fault detection capability (*FDC*) and average execution frequency (*AEF*), and minimizing overall

execution time (*OET*). More specifically, 1) *TMP* measures the number of test cases that can be minimized as compared to the original test suite; 2) *FPC* measures how many pairs of features can be covered by the minimized test cases; 3) *FDC* measures the faults found by the minimized test cases within a specified time period (e.g., one week in the past); 4) *AEF* measures the average execution frequency of the minimized test cases during a given time period; and 5) *OET* measures the time duration for executing the minimized test cases. The *AEF* of a solution is: $AEF = \frac{\sum_{i=1}^{n} EF_{tci}}{n}$, where, n is the number of test cases in the minimized test suite, and EF_{tci} is the execution frequency for the test case tc_i in a given time period (e.g., a week). The objectives are explained in detail in [2]. Similarly, for *TCP*, we defined four objectives: maximizing *FPC*, *FDC* and *AEF*, and minimizing *OET*. Finally, for *TCS*, we defined four objectives: maximizing *FPC*, *FDC,* and *AEF,* and minimizing *TD*, where *TD* measures the difference between the execution time of the selected test cases and the time budget available for testing [17] as shown in Table C-2.

Table C-2. Overview of the Experiment Design*

Domain	*D*	Problem	Objectives	Test Cases	Faults	Sourc Kloc	Test Kloc	Tests
Telecommunication	D_1	*TSM*	*TMP, FPC, FDC, AEF, OET*	489	N/A			
		TCP	*FPC, FDC, OET, AEF*					
		TCS	*FPC, FDC, AEF, TD*					
	D_2	*TSM*	*FDC, SC, CC, TMP, APIC*	211				
		TCP	*FDC, SC, CC, APIC*					
		TCS	*FDC, SC, TD, CC, APIC*					
Maritime	D_3	*TSM*	*MPO, MC, TMP, MPR*	165				
		TCP	*MPO, MC, OET, MPR*					
		TCS	*MPO, MC, TD, MPR*					
Open Source	D_4	*TSM*	*STC, FDC, OET, TMP*	302	26	96	50	2205
		TCP	*STC, FDC, OET*					
		TCS	*SC, FDC, TD*					
	D_5	*TSM*	*STC, FDC, OET, TMP*	125	27	28	53	4130
		TCP	*STC, FDC, OET*					
		TCS	*STC, FDC, TD*					
	D_6	*TSM*	*STC, FDC, OET, TMP*	113	65	22	6	2245
		TCP	*STC, FDC, OET*					
		TCS	*STC, FDC, TD*					
	D_7	*TRA*	*R, C, TR*	N/A				
	D_8	*ITO*	*A, O, R, P*					

**TMP*: test minimization percentage, *FPC*: feature pairwise coverage, *FDC*: fault detection capability, *AEF*: average execution frequency, *OET*: overall execution time, *TD*: time difference, *SC*: status coverage, *CC*: configuration coverage, *APIC*: API coverage, *MPO*: mean probability, *MC*: Mean consequence, *MPR*: mean priority, *STC*: Statement coverage, *R*: Reliability, C: Cost, *TR*: Testing resource expenditure, *A*: number of attributes, *O*: number of operations, *R*: number of distinct return types, *P*: number of distinct parameter types.

2) D_2: D_2 includes 211 test cases with four identified objectives for *TCP* from [3]: maximizing configuration coverage (*CC*), API coverage (*APIC*), status coverage (*SC*), and fault detection capability (*FDC*) as shown in Table C-2. Precisely, 1) *CC* measures the coverage of configuration variables and their values; 2) *APIC* measures the overall

coverage of test API commands, their parameters and their values; 3) *SC* measures the coverage of status variables and their values; and 4) *FDC* measures the number of detected faults within a specified time period (e.g., one month in the past). Detailed descriptions and the mathematical formulae of the objectives are provided in [3].

Additionally, for *TSM*, we defined five objectives in D_2: maximizing *FDC*, *SC*, *CC*, *TMP,* and *APIC*. Finally, for *TCS*, we defined five objectives: maximizing *FDC*, *SC*, *CC,* and *APIC*, and minimizing *TD* (Table C-2).

4.2 Real World Subject (D_3)

The real world subject (i.e., D_3) is from the maritime domain, and it consists of 165 high-level test cases (Table C-2) for testing some of the key elements of subsea oil and gas production systems [17]. The subject was created using different standards (e.g., design and operation of subsea production systems-ISO 13628-6:2006 [45]), OREDA Offshore Reliability Data Handbook [46], and requirements from different oil and gas companies publicly available.

D_3 focused on test case selection (*TCS*) problem within a time budget in [17]. For selecting test cases, we defined four objectives: maximizing mean priority (*MPR*), mean probability (*MPO*), mean consequence (*MC*), and minimizing time difference (*TD*) as shown in Table C-2. For the selected test cases, 1) *MPR* measures the average importance of the test cases based on the type of requirement the test cases check; 2) *MPO* measures the average likelihood that the test cases might find faults; 3) *MC* measures the average impact of failures of the test cases that the system can have on the environment once it is operational; and 4) *TD* measures the difference between the execution time of the selected test cases and the time budget available for testing. Moreover, for *TSM*, we defined four objectives for D_3: maximizing *MPO*, *MC*, *TMP,* and *MPR* (Table C-2). Finally, for *TCP,* we defined four objectives: maximizing *MPO*, *MC* and *MPR*, and minimizing *OET* (Table C-2).

4.3 Open Source Subjects ($D_4 – D_8$)

We chose three subjects from DEFECTS4J [44], one from [4, 5], and one from [25, 35].

4.3.1 Subjects from DEFECTS4J [44] ($D_4 – D_6$)

We chose three programs from the open source projects: JFreeChart[1], Joda-Time[2], and Apache Commons Lang[3] from the DEFECTS4J (v1.1.0) [44] for all three problems: *TSM*, *TCP*, and *TCS*. The details of the three programs are provided in Table C-2.

In Table C-2, the term *test case* refers to each JUnit class in the corresponding programs (e.g., JFreeChart) and the term *test* refers to each test method in the JUnit class as defined

[1] http://jfree.org/jfreechart/
[2] http://joda.org/joda-time/
[3] http://commons.apache.org/lang

in the JUnit framework [47]. The tests verify different functionalities of the programs. For each fault, DEFECTS4J provides the information about the triggering tests (i.e., the test that failed when there was a problem with the particular functionality). Based on this, we can observe how many times the test failed. Therefore, we can count the number of times the test case failed based on the execution history of each test. Note that we have checked the root cause of the failing test cases manually to ensure that all the failing test cases were triggered by actual faults. The open source programs JFreeChart, Joda-Time, and Commons Lang are referred to as D_4, D_5 and D_6, respectively in Table C-2. We have applied *TSM*, *TCP*, and *TCS* for each D_4, D_5 and D_6.

We defined four objectives for *TSM*. The objectives are maximizing statement coverage (*STC*), fault detection capability (*FDC*), test minimization percentage (*TMP*), and minimizing overall execution time (*OET*). For *FDC* we measure the number of times the test case found faults out of the total detected faults in Table C-2. Additionally, for *TCP*, we defined three objectives: maximizing *STC* and *FDC*, and minimizing *OET*. Finally, we defined three objectives for *TCS*: maximizing *STC* and *FDC*, and minimizing *TD* (Table C-2).

4.3.2 Subject from [4, 5] (D_7)

The subject from [4, 5] is used to tackle the testing resource allocation (*TRA*) problem (D_7). The subject consists of eight modules, and the maximum testing resource is set as 10,000 hours as in [4, 5]. In previous work [5], three objectives were identified for the *TRA* problem: maximize reliability (*R*), minimize cost (*C*) and minimize testing resource expenditure (*TR*). The reliability of the module is calculated based on the failure intensity of the model and allocated testing time, the cost is associated with the cost required to achieve the reliability of the module, and total testing resource measures the total allocated testing time for achieving the particular reliability of the systems. The details of the subject can be checked from [5].

4.3.3 Subject from BCEL (D_8)

An open source program "Commons Byte Code Engineering Library (BCEL)[4] version 5.0" was used in [25, 35] for integration and test order (*ITO*) problem. BCEL enables users to analyze, create and manipulate binary Java class files and includes 45 Java classes and 289 dependencies. For the *ITO* problem, four objectives were used as identified in the prior work [25, 35]: number of attributes (*A*), number of operations (*O*), number of distinct return types (*R*), and number of distinct parameter types (*P*).

More specifically, 1) *A* counts the maximum number of attributes that need to be handled in the stub if the dependency is broken; 2) *O* counts the number of operations that need to be emulated if the dependency is broken; 3) *R* counts the number of distinct return types (except the return type void) of the operations that need to be emulated if the

[4] http://archive.apache.org/dist/jakarta/bcel/old/v5.0/

dependency is broken; and 4) *P* counts the number of distinct parameters of the operations that need to be emulated if the dependency is broken. A detailed description of the subject can be checked from [25].

We used different objectives for different subjects such as *FPC* and *AEF* for D_1 based on our collaboration with the industrial partner and the test engineers at the industrial partner valued these objectives for different test optimization problems. However, it is very challenging to get identical information from the subjects from open source projects. Therefore, we used different objectives, such as *STC*, which is used quite often in literature for regression test optimization [8, 48, 49]. Additionally, testing at the industrial partner does not involve subjects as in DEFECTS4J; for example, testing is focused on different variants of video conferencing systems. Moreover, it is not possible to obtain similar information from the industrial partner, and the other two open source subjects (i.e., D_7 and D_8) also do not list them out. Therefore, we only provided information about different programs in DEFECTS4J in Table C-2.

5 Empirical Study Design

This section presents the detailed design of the empirical study (Table C-3), which includes: research questions (Section 5.1), evaluation metrics (Section 5.2), and statistical tests and experiment settings (Section 5.3).

Table C-3. An Overview of the Experiment Design

RQ	Task	Comparison	Subject	Problem	Evaluation Metrics	Statistical Tests
1	$T_{1.1}$	CBGA-ES$^+$ with RS and Greedy	D_1-D_8	*TSM, TCP, TCS, TRA, ITO*	*HV*	Vargha and Delaney, Mann-Whitney U Test
	$T_{1.2}$	CBGA-ES$^+$ with CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2			*HV*, *GS*, *GD*	
2	T_2				Mean fitness values per objective for solutions in the same region	N/A
3	$T_{3.1}$		D_4	*TSM*	*FD*	Vargha and Delaney, Mann-Whitney U Test
	$T_{3.2}$		D_5	*TCP*	*APFD*	
	$T_{3.3}$		D_6	*TCS*	*FD*	
4	$T_{4.1}$		D_1-D_8	*TSM, TCP, TCS, TRA, ITO*	N/A	N/A
	$T_{4.2}$	CBGA-ES$^+$ with CBGA-ES			*HV*	Vargha and Delaney, Mann-Whitney U Test

5.1 Research Questions

RQ1. Sanity Check and comparison with the existing multi-objective search algorithms: This research question is further decomposed into two sub-questions:
RQ1.1. Sanity Check: Is CBGA-ES$^+$ effective as compared with RS and Greedy for addressing the five multi-objective test optimization problems?

If CBGA-ES$^+$ outperforms RS and Greedy, the next step is to compare CBGA-ES$^+$ with its predecessor CBGA-ES and a selected set of search algorithms that have performed well according to the literature [2, 3, 5, 17, 25].

RQ1.2. Comparison with the existing multi-objective search algorithms and predecessor CBGA-ES: Can CBGA-ES$^+$ outperform the selected multi-objective search algorithms (i.e., MOCell, NSGA-II, PAES, and SPEA2) and its predecessor CBGA-ES for addressing the five multi-objective test optimization problems? If CBGA-ES$^+$ performs better than the selected algorithms and CBGA-ES, we have to compute the extent of improvement, which motivates RQ2.

RQ2. Extent of improvement as compared with CBGA-ES and the existing multi-objective search algorithms: To what extent, can CBGA-ES$^+$ improve the performance when comparing with CBGA-ES and the selected four search algorithms in terms of the objectives for addressing each multi-objective test optimization problem for the solutions in the same search space?

RQ3. Performance in terms of fault detection: Does CBGA-ES$^+$ has a better performance than CBGA-ES and the selected multi-objective search algorithms in terms of fault detection?

RQ4. Running time analysis of CBGA-ES$^+$ and the selected multi-objective search algorithms: Is there a significant difference in the running time of CBGA-ES$^+$ as compared to the selected search algorithms? This is due to the fact that it would be infeasible to apply CBGA-ES$^+$ in practice if CBGA-ES$^+$ requires significantly more time to run (e.g., in seconds) than the selected algorithms or its predecessor CBGA-ES. Additionally, we run CBGA-ES$^+$ and CBGA-ES for a fixed time to check if CBGA-ES$^+$ can still perform better than CBGA-ES.

5.2 Evaluation Metrics

It is common to apply quality indicators such as *hypervolume* (*HV*) to compare the overall performance of multi-objective search algorithms [50, 51]. Therefore, for addressing RQ1 using tasks $T_{1.1}$ and $T_{1.2}$ (Table C-3), we used *HV* as an evaluation metric to compare the performance of CBGA-ES$^+$ with RS, Greedy, and the five selected search algorithms based on the guidelines provided in [29]. Specifically, *HV* calculates the volume in the objective space covered by a non-dominated set of solutions (e.g., Pareto front), which is considered as a combined measurement of both convergence and diversity [52].

Moreover, we used the quality indicator *Generated Spread* (*GS*) to measure the diversity of the algorithms and the quality indicator *Generational Distance* (*GD*) to measure the convergence of the algorithms using $T_{1.2}$. Specifically, *GS* extends the quality indicator *Spread* (which only works for two-objective problems), and it measures the extent of spread of the solutions produced by the algorithms [52, 53], while *GD* measures how far are the solutions produced by the algorithms from the nearest solutions in the

optimal Pareto front [54]. A higher value of *HV* demonstrates a better performance while a lower value of *GS* indicates that the generated solutions have better distribution and a value of 0 for *GD* indicates that the obtained solutions by a search algorithm are optimal.

Each algorithm was executed 50 times, and each run produced a number of solutions. To address RQ2, we take a mean for each objective for the solutions produced by the search algorithms in the same region of the search space using task T_2. Additionally, for the three open source subjects (i.e., D_4– D_6), it is possible to find the failing test case for the last faulty version of the programs as explained in Section 4.3.1. Therefore, we use this information to evaluate the effectiveness of the search algorithms in detecting the faults. More specifically, we used the metric *Fault Detection* (*FD)* score [55, 56] to determine if the minimized test suite for *TSM* or selected test cases for *TCS* can detect the faults using $T_{3.1}$ and $T_{3.3}$ for RQ3. Similarly, we employ the widely used *Average Percentage of Fault Detected* (*APFD*) metric [48, 57] for the *TCP* problem using $T_{3.2}$. The *APFD* metric has been widely used to measure the effectiveness of the prioritized test cases in terms of detecting the fault [8, 49, 58]. *FD* score can be calculated as: $FD = \frac{\#\ of\ failing\ tests\ included}{\#\ of\ total\ failing\ tests}$.

The *FD* score ranges between 0 (i.e., when no failing test is included) to 1 (i.e., when all the failing tests are included) inclusive with a higher value implying a better performance. *APFD* for a set of prioritized test cases (s_a) can be calculated as:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{u \times v} + \frac{1}{2u}$$

where, u denotes the number of test cases in the s_a, v denotes the number of faults detected by s_a, and TF_i represents the first test case in s_a that detects the fault i. A higher value of *APFD* implies a better fault detection rate. Finally, we compared the performance of CBGA-ES$^+$ with CBGA-ES by running both of them for a fixed time (10 seconds) for RQ4 using task $T_{4.2}$ as shown in Table C-3.

5.3 Statistical Tests and Experiment Settings

5.3.1 Statistical Tests

Based on the guidelines in [59], the Vargha and Delaney $\hat{A}_{12}$ statistics [60] and Mann-Whitney U test [61] are used to statistically evaluate the results for RQ1, RQ3, and RQ4 as depicted in Table C-3. The Vargha and Delaney statistics is defined as a non-parametric effect size measure and evaluates the probability of yielding higher values for each objective and *HV* for two algorithms *A* and *B*. Additionally, Mann-Whitney U test is used to indicate whether the observations (e.g., objective values) in one data sample are likely to be larger than the observations in another sample, and *p*-value was used to check if the result is significant. We considered a *p*-value below 0.05 as statistically significant, which is a commonly used threshold in SBSE studies [59]. For two algorithms *A* and *B*, *A* has significantly better performance than *B* if $\hat{A}_{12}$ is higher than 0.5 and the *p*-value is less than

0.05. Notice that, we use the Vargha and Delaney comparison without transformation [62] since we are interested in any improvement for the performance of search algorithms [63, 64].

5.3.2 Experiment Settings

We implemented CBGA-ES$^+$ using jMetal [65] since it integrates most of the existing search algorithms (e.g., SPEA2) and random search (RS), and it has been widely applied (e.g., [66-68]). Note that RS is treated as a multi-objective variant in jMetal. Moreover, the selected search algorithms together with *HV, GS,* and *GD* were implemented from jMetal, and we also encoded the eight subjects there. We used the same *population size* in all the search algorithms, which is the standard settings for configuring the search algorithms in jMetal, and we tuned the additional parameters of the six search algorithms across the eight subjects using the iRace optimization package [69]. Table C-4 presents the parameter settings of the search algorithms across the eight subjects. Additionally, we set the maximum number of fitness evaluations (i.e., termination criteria) as 50,000 for all the algorithms across the eight subjects.

Table C-4. Parameter Settings of the Search Algorithms*

Algorithm	Parameter Settings	PM	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
NSGA-II	***Population Size***: 100; ***Selection of Parents***: binary tournament + binary tournament; ***Recombination***: simulated binary; ***Mutation***: swap	CR	0.93	0.87	0.94	0.93	0.84	0.26	0.95	0.39
		MR	0.59	0.01	0.01	0.02	0.01	0.05	0.13	0.04
SPEA2	***Population Size***: 100; ***Selection of Parents***: binary tournament + binary tournament; ***Recombination***: simulated binary; ***Mutation***: swap; ***Archive size***: 100	CR	0.96	0.95	0.99	0.71	0.76	0.62	0.85	0.94
		MR	0.01	0.01	0.01	0.01	0.01	0.04	0.04	0.02
MOCell	***Population Size***: 100; ***Neighborhood***: 1-hop neighbors (8 surrounding solutions); **Selection of Parents**: binary tournament + binary tournament; ***Recombination:*** simulated binary; ***Mutation***: swap; ***Archive size***: 100	CR	0.70	0.60	0.40	0.84	0.91	0.61	0.88	0.97
		MR	0.64	0.01	0.01	0.01	0.01	0.06	0.03	0.04
PAES	***Mutation***: swap; ***Archive size***: 100	MR	0.02	0.03	0.01	0.03	0.01	0.04	0.60	0.03
CBGA-ES	***Population Size***: 100; ***Recombination***: simulated binary; ***Mutation***: swap	CR	0.73	0.61	0.62	0.67	0.91	0.78	0.74	0.97
		MR	0.02	0.01	0.03	0.03	0.02	0.02	0.04	0.02
		NC	4	18	7	6	8	6	6	10
		EP	76	71	63	57	92	84	81	96
CBGA-ES$^+$	***Population Size***: 100; ***Recombination***: simulated binary; ***Mutation***: swap	CR	0.26	0.97	0.16	0.63	0.83	0.88	0.54	0.84
		MR	0.01	0.01	0.02	0.02	0.03	0.03	0.06	0.01
		NC	9	16	6	4	9	6	9	6
		EP	43	81	77	82	56	90	95	97

* PM: Parameter, CR: Crossover Rate, MR: Mutation Rate, NC: Number of Cluster, EP: Size of Elite Population.

As for the subjects (Section 4), we encoded the test suites in D_1 - D_6 for *TSM*, *TCP*, and *TCS*, the modules in D_7 for *TRA*, and the classes and the dependencies between them for *ITO* in D_8 as an abstract format, which contains the key information of the entity (e.g., test cases) for optimization, e.g., 1) test case *id*, historical execution time for each test case for D_1 - D_6, 2) module *id*, constants for each module for D_7, and 3) class *id*, dependent class (es) *id* for each class for D_8. Notice that the test cases, modules, and classes in the abstract

format can be easily mapped to the original test suite, module list, and class list, respectively using their *ids*. Afterward, the search algorithms (e.g., CBGA-ES$^+$) are employed to produce the optimized solutions (e.g., a set of prioritized test cases for *TCP*) formed as the same abstract format, which consists of a list of optimized entities. Last, the optimized entities are selected for their respective purpose. For example, the test cases are selected from the original test suite using the test case *id* and put for execution for the *TCS* problem.

Additionally, we used the open source tool EclEmma [70] to measure the statement coverage for D_4, D_5, and D_6. Each algorithm was executed 50 times for each subjectto account for the random variation of search algorithms [59]. All the experiments were conducted on the Abel supercomputer at the University of Oslo [71].

Based on the eight subjects and five multi-objective test optimization problems (defined in Section 4), 20 experiments were performed in total for each optimization algorithm. Specifically, 18 experiments were performed with six subjects (D_1 - D_6) for three test optimization problems (*TSM*, *TCP*, and *TCS*), and two experiments were performed with two subjects (D_7 and D_8) for the remaining two test optimization problems (*TRA* and *ITO*). All the optimization problems were not applied to each subject because *TSM*, *TCP*, and *TCS* require subjects with test cases, while *TRA* and *ITO* require subjects with different units (e.g., classes). Thus, the total number of valid experiments is 20 for each optimization algorithm.

6 Results and Analysis

6.1 RQ1. Sanity Check and Comparison with the Selected Search Algorithms

6.1.1 RQ1.1. Sanity Check

In RQ1.1, CBGA-ES$^+$ is compared with Greedy and RS in terms of *HV* for the eight subjects, such that all the values from the algorithms in 50 runs are considered for the comparison. Using the Vargha and Delaney $\hat{A}_{12}$ statistics and Mann-Whitney U test to analyze the results, we observed that CBGA-ES$^+$ significantly outperformed: RS for 100% (i.e., 20 out of 20) of the experiments and Greedy for 85% (i.e., 17 out of 20) of the experiments since $\hat{A}_{12}$ for all the experiments is greater than 0.9, and p-value is less than 0.05.

6.1.2 RQ1.2. Comparison with the Selected Search Algorithms

This section aims to answer RQ1.2 by comparing the performance of CBGA-ES$^+$ with the five selected search algorithms: MOCell, NSGA-II, PAES, SPEA2, and CBGA-ES.

TSM. On employing the different quality indicators, it can be observed from Table C-5, that CBGA-ES$^+$ achieved significantly higher values of *HV* for 60% (i.e., 18 out of 30) of the cases while there was no significant difference in the performance for an average of 6.7% (i.e., 2 out of 30) of the cases. Regarding *GS*, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 16.7% (i.e., 5 out of 30) of the cases while there was no significant difference in the performance for an average of 13.3% (i.e., 4 out of 30) of the cases. Concerning *GD*, CBGA-ES$^+$ achieved a significantly better performance for an average of 90% (i.e., 27 out of 30) of the cases, while there was no significant difference in the performance for an average of 3.3% of the cases as shown in Table C-5.

Table C-5. Quality Indicators compared to CBGA-ES$^+$ for *TSM**

D	**Compared with**	***HV***		***GS***		***GD***	
		$\widehat{A}_{12}$	***p*-value**	$\widehat{A}_{12}$	***p*-value**	$\widehat{A}_{12}$	***p*-value**
D_1	CBGA-ES	0.71	**<0.05**	0.98	**<0.05**	0.16	**<0.05**
	MOCell	0.77	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.79	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.98	**<0.05**	1.00	**<0.05**
	SPEA2	0.66	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_2	CBGA-ES	0.91	**<0.05**	0.99	**<0.05**	0.99	**<0.05**
	MOCell	0.79	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.78	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	1.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.75	**<0.05**	0.00	**<0.05**	0.95	**<0.05**
D_3	CBGA-ES	0.58	**<0.05**	0.44	**<0.05**	0.60	**<0.05**
	MOCell	0.80	**<0.05**	0.50	1.00	0.80	**<0.05**
	NSGA-II	0.51	0.33	0.50	1.00	0.51	0.33
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.71	**<0.05**	0.50	1.00	0.71	**<0.05**
D_4	CBGA-ES	0.58	0.16	0.00	**<0.05**	1.00	**<0.05**
	MOCell	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_5	CBGA-ES	1.00	**<0.05**	0.41	0.12	1.00	**<0.05**
	MOCell	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_6	CBGA-ES	0.67	**<0.05**	0.95	**<0.05**	0.01	**<0.05**
	MOCell	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.02	**<0.05**	0.93	**<0.05**
	SPEA2	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**

**D*: Subject; the bold numbers in the table imply that the results are statistically significant.

TCP. It can be observed from Table C-6 that in terms of *HV*, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 66.7% (i.e., 20 out of 30) of the

cases. However, in terms of *GD*, CBGA-ES$^+$ performed better than the selected algorithms for only 3.3% of the cases while there was no significant difference in the performance for an average of 6.7% (i.e., 2 out of 30) of the cases. Contrarily, with respect to *GD*, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 96.7% (i.e., 29 out of 30) of the cases (Table C-6).

Table C-6. Quality Indicators compared to CBGA-ES$^+$ for *TCP**

D	**Compared with**	***HV***		***GS***		***GD***	
		$\widehat{A}_{12}$	***p*-value**	$\widehat{A}_{12}$	***p*-value**	$\widehat{A}_{12}$	***p*-value**
D_1	CBGA-ES	0.80	**<0.05**	0.46	0.46	0.74	**<0.05**
	MOCell	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.05	**<0.05**	1.00	**<0.05**
	SPEA2	0.62	**<0.05**	0.00	**<0.05**	0.86	**<0.05**
D_2	CBGA-ES	0.73	**<0.05**	0.59	0.13	0.88	**<0.05**
	MOCell	0.93	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.69	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.71	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_3	CBGA-ES	0.69	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	MOCell	0.75	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.64	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.95	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_4	CBGA-ES	1.00	**<0.05**	0.03	**<0.05**	1.00	**<0.05**
	MOCell	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_5	CBGA-ES	0.33	**<0.05**	0.76	**<0.05**	0.83	0.35
	MOCell	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_6	CBGA-ES	0.63	**<0.05**	0.11	**<0.05**	1.00	**<0.05**
	MOCell	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**

**D*: Subject; the bold numbers in the table imply that the results are statistically significant.

***TCS*.** Regarding *HV*, it can be observed from Table C-7 that CBGA-ES$^+$ performed significantly better than the selected algorithms for an average of 80.0% (i.e., 24 out of 30) of the cases, while there was no significant difference in the performance for an average of 13.3% (i.e., 4 out of 30) of the cases. Additionally, regarding *GS*, CBGA-ES$^+$ significantly outperformed the selected algorithms for 6.7% of the cases, while there was no significant difference in the performance for 13.3% (i.e., 4 out of 30) of the cases (Table C-7). Finally, regarding *GD*, CBGA-ES$^+$ achieved a better performance than the selected algorithms for an average of 76.7% (i.e., 23 out of 30) of the cases, while there was no significant difference in the performance for 10.3%.

Table C-7. Quality Indicators compared to CBGA-ES$^+$ for *TCS**

Subject	Compared with	*HV*		*GS*		*GD*	
		$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value
D_1	CBGA-ES	0.67	**<0.05**	0.00	**<0.05**	0.97	**<0.05**
	MOCell	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	0.99	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
D_2	CBGA-ES	0.54	0.48	0.60	0.09	1.00	**<0.05**
	MOCell	0.71	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.66	**<0.05**	0.00	**<0.05**	0.99	**<0.05**
	PAES	0.82	**<0.05**	0.00	**<0.05**	0.99	**<0.05**
	SPEA2	0.65	**<0.05**	0.00	**<0.05**	0.96	**<0.05**
D_3	CBGA-ES	0.66	**<0.05**	0.44	0.29	0.55	0.35
	MOCell	0.81	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	NSGA-II	0.66	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
	SPEA2	1.00	**<0.05**	0.00	**<0.05**	0.99	**<0.05**
D_4	CBGA-ES	1.00	**<0.05**	0.12	**<0.05**	0.77	**<0.05**
	MOCell	0.51	0.93	0.27	**<0.05**	0.37	**<0.05**
	NSGA-II	0.38	**<0.05**	0.07	**<0.05**	0.80	**<0.05**
	PAES	1.00	**<0.05**	0.02	**<0.05**	0.99	**<0.05**
	SPEA2	0.66	**<0.05**	0.13	**<0.05**	0.52	0.68
D_5	CBGA-ES	0.93	**<0.05**	0.37	**<0.05**	0.98	**<0.05**
	MOCell	0.44	0.27	0.65	**<0.05**	0.12	**<0.05**
	NSGA-II	0.36	**<0.05**	0.75	**<0.05**	0.03	**<0.05**
	PAES	1.00	**<0.05**	0.17	**<0.05**	0.99	**<0.05**
	SPEA2	0.46	0.54	0.57	0.26	0.02	**<0.05**
D_6	CBGA-ES	1.00	**<0.05**	0.41	0.14	1.00	**<0.05**
	MOCell	0.95	**<0.05**	0.17	**<0.05**	0.74	**<0.05**
	NSGA-II	0.73	**<0.05**	0.20	**<0.05**	0.82	**<0.05**
	PAES	1.00	**<0.05**	0.20	**<0.05**	0.47	0.57
	SPEA2	0.76	**<0.05**	0.17	**<0.05**	0.61	**<0.05**

*the bold numbers n the table imply that the results are statistically significant.

***TRA* and *ITO*.** As shown in Table C-8, for *TRA*, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 20% (i.e., 1 out of 5) of the cases and showed no difference for an average of 20% of the cases in terms of *HV*. Moreover, in terms of *GS*, CBGA-ES$^+$ performed better than the selected algorithms for an average of only 20% (i.e., 1 out of 5) of the cases. On the contrary, with respect to *GD*, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 100% of the cases.

Table C-8. Quality Indicators compared to CBGA-ES$^+$ for *TRA* and *ITO**

Problem	Subject	Compared with	*HV*		*GS*		*GD*	
			$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value
TRA	D_7	CBGA-ES	0.44	0.29	0.69	**<0.05**	0.82	**<0.05**
		MOCell	0.04	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
		NSGA-II	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
		PAES	1.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
		SPEA2	0.00	**<0.05**	0.00	**<0.05**	1.00	**<0.05**
ITO	D_8	CBGA-ES	0.26	**<0.05**	1.00	**<0.05**	0.97	**<0.05**

		MOCell	0.69	**<0.05**	0.59	**0.02**	0.59	0.13
		NSGA-II	0.64	**<0.05**	0.68	**<0.05**	0.68	**<0.05**
		PAES	0.98	**<0.05**	1.00	**<0.05**	1.00	**<0.05**
		SPEA2	0.61	0.05	0.63	**<0.05**	0.63	**<0.05**

*the bold numbers in the table imply that the results are statistically significant.

Additionally, for ***ITO***, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 60% (i.e., 3 out of 5) of the cases and showed no significant difference for 20% of the cases in terms of *HV*. Moreover, in terms of *GS*, CBGA-ES$^+$ performed significantly better than the selected algorithms for all the cases. Finally, with respect to *GD*, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 80% of the cases.

Concluding Remarks. We can answer RQ1 as: CBGA-ES$^+$ can significantly outperform the baseline algorithms (i.e., RS and Greedy) and the five selected algorithms for the majority of the five test optimization problems. Overall, CBGA-ES$^+$ managed to significantly outperform RS for 100% (i.e., 20 out of 20) of the experiments and Greedy for 85% of the experiments in terms of *HV*. Moreover, as compared to the five selected algorithms, CBGA-ES$^+$ produced better overall quality solutions (as indicated by *HV*) for an average of 66.0% (66 out of 100) of the cases (i.e., 5 selected search algorithms × 20 experiments) while there were no significant differences for 8.0% (8 out of 100) of the cases, 2) CBGA-ES$^+$ had significantly higher spread (as indicated by *GS*) for an average of 14.0% (14 out of 100) of the cases while there were no significant differences for 10.0% (10 out of 100) of the cases, and 3) CBGA-ES$^+$ managed to obtain significantly better quality solutions, i.e., close to the optimal Pareto front (as indicated by *GD*) for 88.0% (88 out of 100) of the cases while there were no significant differences for 6.0% (6 out of 100) of the cases.

6.2 RQ2. Extent of Improvement

This research question aims to answer to what extent CBGA-ES$^+$ can improve (i.e., perform better than) its predecessor CBGA-ES and the existing search algorithms (i.e., MOCell, NSGA-II, PAES, and SPEA2) in terms of each objective for the eight subjects (defined in Section 4) for solutions within the same regions of the search space. In our context, a search space region is defined by specifying a lower and/or an upper bound for each objective as done in [72]. First, a reference front is created by combining the non-dominated solutions from all the selected algorithms. Second, some solutions are selected from the reference front, such that the selected solutions have the highest average values of all the defined objectives (by using a scalar objective function and providing equal weights to each objective). Third, a neighborhood is chosen and expressed in terms of an acceptable percentage variation in the objective value, e.g., 10% to ensure that all the selected algorithms have solutions in the selected search space. Fig. C-3 shows the percentage, by which CBGA-ES$^+$ improved or deteriorated the performance as compared

to CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2 for each objective in the eight subjects (i.e., D_1 –D_8) for the five multi-objective test optimization problems, i.e., a total of 20 experiments.

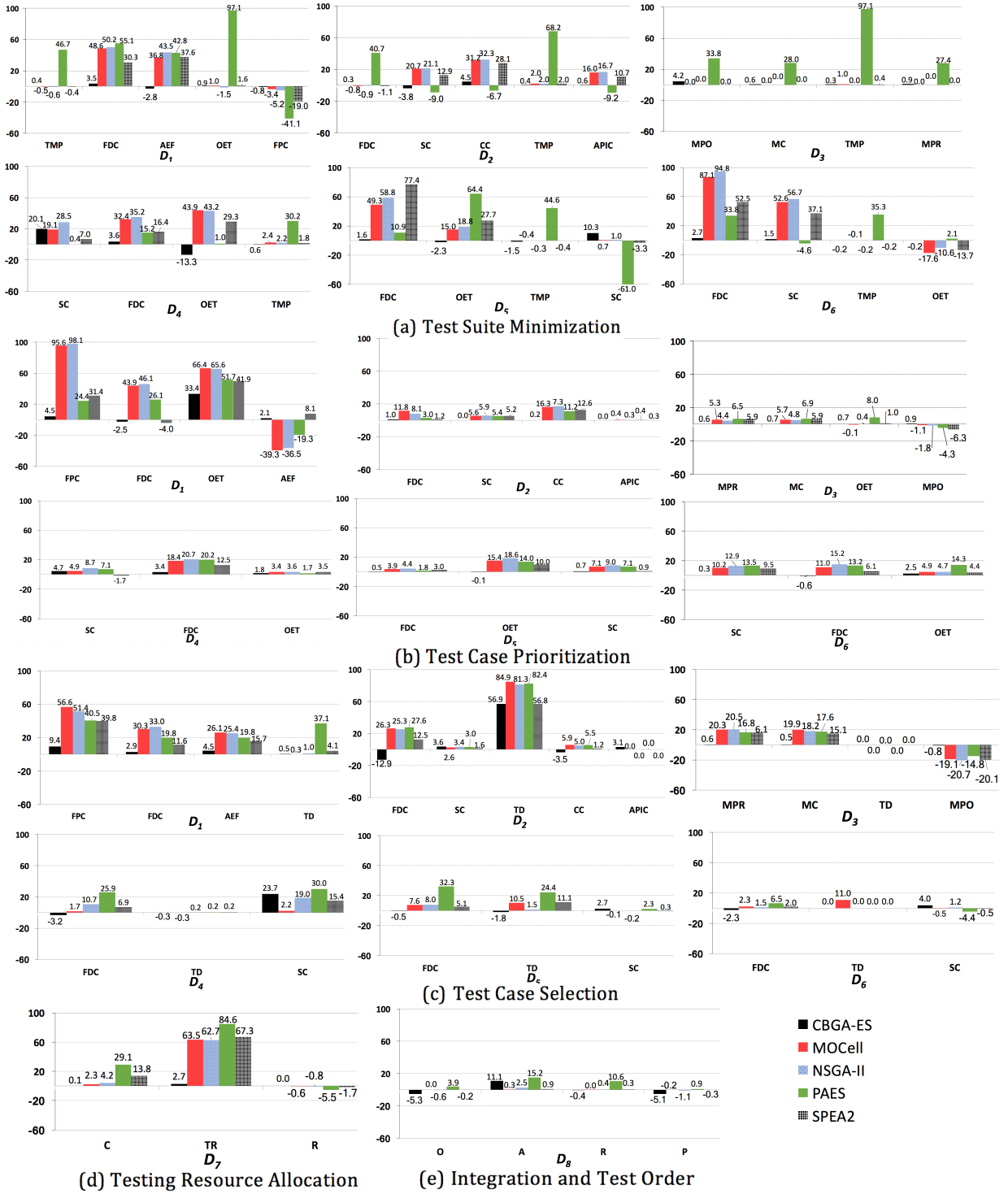

Fig. C-3. Percentage of objectives CBGA-ES$^+$ is better/worse than CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2

Table C-9 summarizes the improvement of CBGA-ES$^+$ with respect to CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2 for the five multi-objective test optimization problems with the eight subjects. For a specific subject focusing a particular problem, the average improvement of CBGA-ES$^+$ against a particular search algorithm is calculated using the formula: $Average\ improvement = \frac{\sum_{i=1}^{n} per_i}{n}$, where, n is the number of objectives in the subject, and per_i is the percentage by which CBGA-ES$^+$ performed better/worse than a particular search algorithm for the objective i in the subject. For instance, from Fig. C-3, for D_1 in the *TSM* problem, as compared to MOCell, CBGA-ES$^+$ had an average improvement of:

$$\frac{-0.5\% + 48.6\% + 36.8\% + 1.0\% - 3.4\%}{5} = 16.5\%$$

Table C-9. Average Improvement Percentages Achieved by CBGA-ES$^+$

Problem	Subject	CBGA-ES	MOCell	NSGA-II	PAES	SPEA2
TSM	D_1	0.2	16.5	17.3	40.1	10.0
	D_2	0.4	13.8	14.2	16.8	10.5
	D_3	1.5	0.3	0.0	46.6	0.1
	D_4	2.7	24.5	27.3	11.7	13.6
	D_5	2.0	16.2	19.5	14.7	25.4
	D_6	0.9	30.5	35.2	16.7	18.9
TCP	D_1	9.4	41.7	43.3	20.7	21.4
	D_2	0.3	8.5	7.9	5.0	4.8
	D_3	0.7	2.4	1.9	4.3	1.6
	D_4	3.3	8.9	11.0	9.7	4.8
	D_5	0.4	8.8	10.7	7.6	4.6
	D_6	0.7	8.7	10.9	13.7	6.7
TCS	D_1	4.3	28.4	27.7	29.3	17.8
	D_2	9.5	23.9	23.0	23.7	14.4
	D_3	0.1	5.3	4.5	4.9	2.8
	D_4	6.8	1.2	10.0	18.7	7.5
	D_5	0.1	6.0	3.1	19.7	5.5
	D_6	0.6	4.2	0.9	0.7	0.5
TRA	D_7	0.9	21.7	22.1	36.1	26.4
ITO	D_8	0.1	0.0	0.3	7.7	0.2
Average		2.2	13.6	14.5	17.4	9.9

Concluding Remarks. We can answer RQ2 as: CBGA-ES$^+$ improves the performance of the majority of the individual objectives to a large extent as compared to the four selected search algorithms and to a smaller extent as compared to CBGA-ES for the solutions in the same region of the search space. In terms of practical implications, we can observe that CBGA-ES$^+$ can generate better quality solutions in terms of the defined objectives, and thus improve the overall quality of testing.

6.3 RQ3. Performance in Terms of Fault Detection

Since the search algorithms are multi-objective in nature, each run produces many Pareto efficient solutions (solutions with similar quality) as described in Section 5.3.2. In this section, we evaluate the performance of CBGA-ES$^+$ in terms of fault detection for *TSM*, *TCP*, and *TCS*.

6.3.1 TSM

Table C-10 shows the result of comparing CBGA-ES+ with the selected five search algorithms in terms of *FD* using $D_4 - D_6$. Based on Table C-10, it can be observed that *FD* scores produced by CBGA-ES+ are significantly higher than the selected algorithms for an average of 80% (i.e., 12 out of 15) of the cases. Moreover, the solutions produced by CBGA-ES+ have a much higher chance to detect the faults, which is on average 17.8% higher (Fig. C-4).

Table C-10. Comparison of *FD* with respect to CBGA-ES+ for *TSM*

Algorithms	D_4		D_5		D_6	
	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value
CBGA-ES	**0.52**	**<0.05**	**0.61**	**<0.05**	**0.59**	**<0.05**
MOCell	**0.72**	**<0.05**	**0.48**	**<0.05**	**0.64**	**<0.05**
NSGA-II	**0.72**	**<0.05**	**0.45**	**<0.05**	**0.66**	**<0.05**
PAES	**0.54**	**<0.05**	**0.28**	**<0.05**	**0.80**	**<0.05**
SPEA2	**0.65**	**<0.05**	**0.53**	**<0.05**	**0.66**	**<0.05**

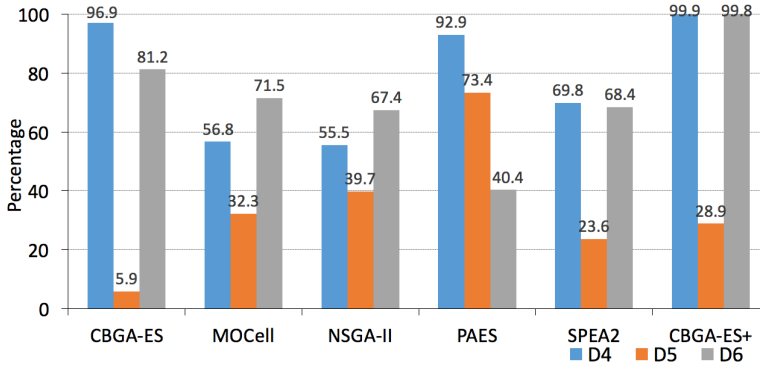

Fig. C-4. Percentage of solutions that detected faults for *TSM*

6.3.2 TCP

As shown in Table C-11, in terms of *APFD*, CBGA-ES+ significantly outperformed the selected algorithms for an average of 86.7% (i.e., 13 out of 15) of the cases, and there was no significant difference in the performance for an average of 6.7% of the cases.

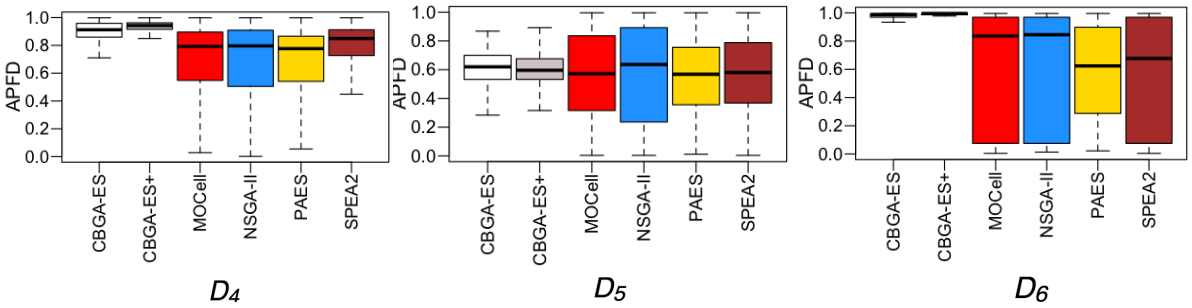

Fig. C-5. *APFD* scores for *TCP* using $D_4 - D_6$

Table C-11. Comparison of *APFD* with respect to CBGA-ES$^+$ for *TCP*

Algorithms	D_4		D_5		D_6	
	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value
CBGA-ES	0.63	**<0.05**	0.46	**<0.05**	0.64	**<0.05**
MOCell	0.86	**<0.05**	0.52	**<0.05**	0.89	**<0.05**
NSGA-II	0.84	**<0.05**	0.50	0.84	0.91	**<0.05**
PAES	0.90	**<0.05**	0.54	**<0.05**	0.96	**<0.05**
SPEA2	0.83	**<0.05**	0.52	**<0.05**	0.91	**<0.05**

Moreover, as shown in Fig. C-5, the solutions produced by CBGA-ES$^+$ have comparatively higher median *APFD* scores as compared to the selected algorithms. Overall, the solutions produced by CBGA-ES$^+$ had a comparatively higher mean *APFD* score of 18.2% on average as compared to the selected algorithms.

6.3.3 TCS

Table C-12 presents the result of comparing CBGA-ES$^+$ with the selected five search algorithms in terms of *FD* using D_4– D_6. As shown in Table C-12, CBGA-ES$^+$ significantly outperformed the selected algorithms for an average of 80% (i.e., 12 out of 15) of the cases while there was no significant difference for 6.7% of the cases.

Table C-12. Comparison of *FD* with respect to CBGA-ES$^+$ for *TCS*

Algorithms	D_4		D_5		D_6	
	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value
CBGA-ES	0.50	0.53	0.62	**<0.05**	0.53	**<0.05**
MOCell	0.54	**<0.05**	0.49	**<0.05**	0.57	**<0.05**
NSGA-II	0.55	**<0.05**	0.49	**<0.05**	0.58	**<0.05**
PAES	0.65	**<0.05**	0.72	**<0.05**	0.68	**<0.05**
SPEA2	0.54	**<0.05**	0.51	**<0.05**	0.57	**<0.05**

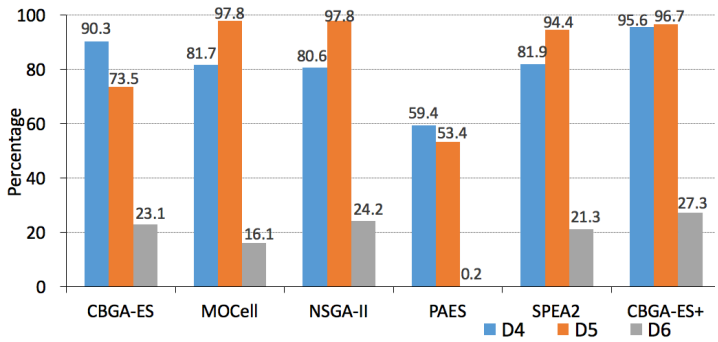

Fig. C-6. Percentage of solutions that detected fault for *TCS*

The solutions produced by CBGA-ES$^+$ have a much higher chance to detect the faults (Fig. C-6), which is on average 13.8% higher than the selected search algorithms.

Concluding Remarks. We can answer RQ3 as CBGA-ES$^+$ significantly outperformed CBGA-ES and the selected four search algorithms in terms of fault detection for most of the experiments. Overall, the solutions produced by managed to detect faults 17.8% and 13.5% higher on average as compared to CBGA-ES and the selected four search algorithms for *TSM* and *TCS*. Moreover, CBGA-ES$^+$ had a comparatively higher mean *APFD* score of 18.2% on average than the selected algorithms.

6.4 RQ4. Time Analysis

In Table C-13 presents the average running time of CBGA-ES$^+$ and the five selected search algorithms for the eight subjects across the five multi-objective optimization problems. Based on the results from Table C-13, we can observe that on average the running time of CBGA-ES$^+$ is quite comparable to the four selected search algorithms and faster than CBGA-ES for all the 20 experiments across the eight subjects.

Since the average improvement of CBGA-ES$^+$ over CBGA-ES is on average 2.2% higher in terms of individual objectives (Table C-9), it is essential to check if the performance of CBGA-ES$^+$ is similar to CBGA-ES by running them for a similar fixed time. Therefore, we compare the performance of CBGA-ES$^+$ with CBGA-ES by running them for a same fixed time (i.e., 10 seconds) for eight subjects as shown in Table C-14. Based on the results from Table C-14, one can observe that CBGA-ES$^+$ significantly outperformed CBGA-ES by 95% (i.e., 19 out of 20) of the experiments.

Table C-13. Average Running Time of the Algorithms

D	**Running Time (in seconds)**					
	CBGA-ES$^+$	**CBGA-ES**	**MOCell**	**NSGA-II**	**PAES**	**SPEA2**
D_1	14.1	16.9	21.4	17.0	33.0	19.8
D_2	55.6	60.9	57.1	58.1	85.6	64.1
D_3	5.2	6.2	5.5	4.3	9.0	7.4
D_4	7.4	10.1	9.2	6.5	7.5	10.2
D_5	4.6	6.4	3.9	3.3	4.5	6.6
D_6	4.8	5.6	4.2	3.2	5.2	6.8
D_7	2.6	3.7	1.9	1.3	1.1	3.4
D_8	4.9	6.5	2.9	3.6	5.2	5.4

Table C-14. Comparison of CBGA-ES$^+$ and CBGA-ES using *HV**

D	***TSM***		***TCP***		***TCS***		***TRA***		***ITO***	
	$\widehat{A}_{12}$	***p*-val**	$\widehat{A}_{12}$	***p*-val**	$\widehat{A}_{12}$	***p*-val**	$\widehat{A}_{12}$	***p*-val**	$\widehat{A}_{12}$	***p*-val**
D_1	0.85	**<0.05**	0.91	**<0.05**	0.81	**<0.05**	N/A			
D_2	0.99	**<0.05**	0.95	**<0.05**	0.99	**<0.05**				
D_3	0.57	**<0.05**	0.70	**<0.05**	0.96	**<0.05**				
D_4	0.99	**<0.05**	0.94	**<0.05**	1.00	**<0.05**				
D_5	1.00	**<0.05**	1.00	**<0.05**	1.00	**<0.05**				
D_6	0.87	**<0.05**	0.88	**<0.05**	1.00	**<0.05**				
D_7	N/A						0.55	0.42	N/A	
D_8	N/A								0.83	**<0.05**

Concluding Remarks. We can conclude that 1) there is no practical difference in terms of running time for CBGA-ES$^+$ as compared with the selected four search algorithms (i.e., MOCell, NSGA-II, PAES, and SPEA2) and 2) CBGA-ES$^+$ is faster than CBGA-ES for all the five multi-objective test optimization problems. Moreover, CBGA-ES$^+$ significantly outperformed CBGA-ES for almost all the problems after running for a fixed time of 10 seconds.

7 Overall Discussion

For RQ1, we observed that CBGA-ES$^+$ significantly outperformed RS for 100% of the experiments with the eight subjects (i.e., $D_1 - D_8$, Section 4) for five multi-objective test optimization problems. This observation suggests that our five multi-objective test optimization problems (i.e., *TSM, TCP, TCS, TRA,* and *ITO*) are not trivial to solve and require an efficient optimization approach. Moreover, CBGA-ES$^+$ performed better than Greedy for 85% of the experiments for the five multi-objective test optimization problems (recall Section 6.1). This can be explained by the fact that Greedy greedily selects the best test case one at a time in terms of defined objectives until the termination conditions are met. Sometimes, Greedy may get stuck in local search space and result in sub-optimal solutions [73]. However, CBGA-ES$^+$ employs *mutation* operator for exploring the global search space with the aim to obtain optimal solutions. Moreover, CBGA-ES$^+$ produces a set of non-dominated solutions for preserving the optimal solutions with equivalent quality as compared to Greedy that might lose optimal solutions holding the same quality [32].

The reason for the better performance of CBGA-ES$^+$ as compared to the four selected search algorithms (i.e., MOCell, NSGA-II, PAES, and SPEA2) can be explained as follows: In each generation, CBGA-ES$^+$ employs the elitist selection and k-means clustering algorithm to cluster and only select the best non-dominated solutions for applying *crossover* and *mutation* operators to generate offspring solutions. Our *cluster dominance strategy* (Section 3.1) helps to distinguish the performance of different clusters even when the clusters do not dominate other completely, which enables us to get elite solutions from a population, which will be used for producing offspring solutions.

More specifically, suppose that the population size of NSGA-II, SPEA2, and MOCell is N and there are in total M elite solutions that exist in the best cluster from the population. M should be less than N since it is practically infeasible that all the solutions from the population are elite. For NSGA-II, SPEA2, and MOCell, the parent solutions will be selected from the entire population, and thus, the probability of selecting two parent solutions can be measured as $\frac{2}{N\times(N-1)}$. However, CBGA-ES$^+$ only selects the parent solutions from the M elite solutions, and therefore the probability of selecting two parent solutions is $\frac{2}{M\times(M-1)}$. We can observe that CBGA-ES$^+$ has less randomness than NSGA-II, SPEA2, and MOCell in terms of selecting two parent solutions.

Moreover, for NSGA-II, SPEA2 and MOCell, the probability of selecting elite solutions as two parent solutions for one reproduction is $\frac{M}{N} \times \frac{M-1}{N-1}$ (which is less than or equal to 1), while CBGA-ES$^+$ can ensure that the elite solutions will be chosen for producing offspring solutions with a 100% probability. As for PAES, two parent solutions are randomly initialized for producing offspring solutions, which still makes the probability of selecting elite solutions as parent solutions less than 100% (since the number of elite solutions should be less than the total number of potential candidate solutions), which is the key reason to explain worse performance of PAES than CBGA-ES$^+$. Furthermore, the performance of CBGA-ES$^+$ is also significantly better than its predecessor CBGA-ES since CBGA-ES$^+$ selects only the non-dominated solutions from the best clusters, unlike CBGA-ES that selects all the best solutions (even the dominated ones) from the best clusters. This allows CBGA-ES$^+$ to select better elite solutions for producing offspring solutions.

CBGA-ES$^+$ was not able to significantly outperform the selected search algorithms for some objectives (e.g., *FPC* in D_1 for *TSM* in Fig. C-3) because it focuses on improving the majority of the objectives, and if the improvement of some objectives might result in a decrease in the performance of the majority of the objectives, it might ignore the improvement of these objectives. The search algorithms maintain a tradeoff between exploration (as indicated by *GS*) and exploitation (as indicated by *GD*) to produce the optimal solutions. CBGA-ES$^+$ favors exploitation and tends to produce solutions that are closer to the optimal Pareto front. The logic for doing so is because normally in practice only one solution is executed from all the generated solutions, which the test engineers select randomly or based on domain expertise since there is no guideline on selecting the best solution. Therefore, all the generated solutions need to be better than those by the other algorithms based on the pre-defined evaluation criteria (e.g., number of fitness evaluations). As indicated by *GD* in Section 6.1.2, the solutions generated by CBGA-ES$^+$ are much closer to the solutions in the optimal Pareto front as compared to the selected algorithms for 88% of the experiments.

Regarding the parameter tuning with iRace, we ran 1000 experiments for each algorithm for each subject. iRace lists out one or more number of best candidate solutions based on the conducted experiments, for each subject, and we selected one of those parameter(s) for each subject in our experiments. To study the influence of fea features of the subject for the algorithm parameters requires another study, which will be conducted in the future.

Moreover, to evaluate whether the solutions produced by CBGA-ES$^+$ can dominate the solutions produced by the selected search algorithms, we compare the non-dominated solutions produced by CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2 with the non-dominated solutions produced by CBGA-ES$^+$ obtained in 50 runs. More specifically, after running the search algorithms 50 times each time to account for randomness, we retain only the non-dominated solutions for CBGA-ES$^+$, CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2. After that, we compare the non-dominated solutions from CBGA-ES$^+$ with CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2.

Fig. C-7 presents the percentage of non-dominated solutions of CBGA-ES, MOCell, NSGA-II, PAES, and SPEA2 that is dominated by CBGA-ES$^+$ and vice versa on average for the five multi-objective test optimization problems. Based on Fig. C-7, it can be observed that CBGA-ES$^+$ dominates on average 24.9%, 51.2%, and 62.4% of the non-dominated solutions produced by the five selected algorithms for *TSM*, *TCP*, and *TCS*. For *TRA*, CBGA-ES$^+$ dominates less than 1% of the non-dominated solutions produced by MOCell, NSGA-II, PAES, and SPEA2 and also has less diversity. This might be the reason for the worse overall quality of solutions for CBGA-ES$^+$ (as indicated by *HV*). For D_8, even though CBGA-ES$^+$ dominates less than 1% of the non-dominated solutions produced by MOCell, NSGA-II, and SPEA2, CBGA-ES$^+$ has both higher convergence and divergence, which is the reason for a higher *HV*. Overall, on average CBGA-ES$^+$ dominated 33.8%, 24.7%, 27.7%, 54.0%, and 18.9% of the non- dominated on average 4.5%, 2.2%, 1.4%, 0.0%, and 1.7% of the non-dominated solutions produced by CBGA-ES$^+$.

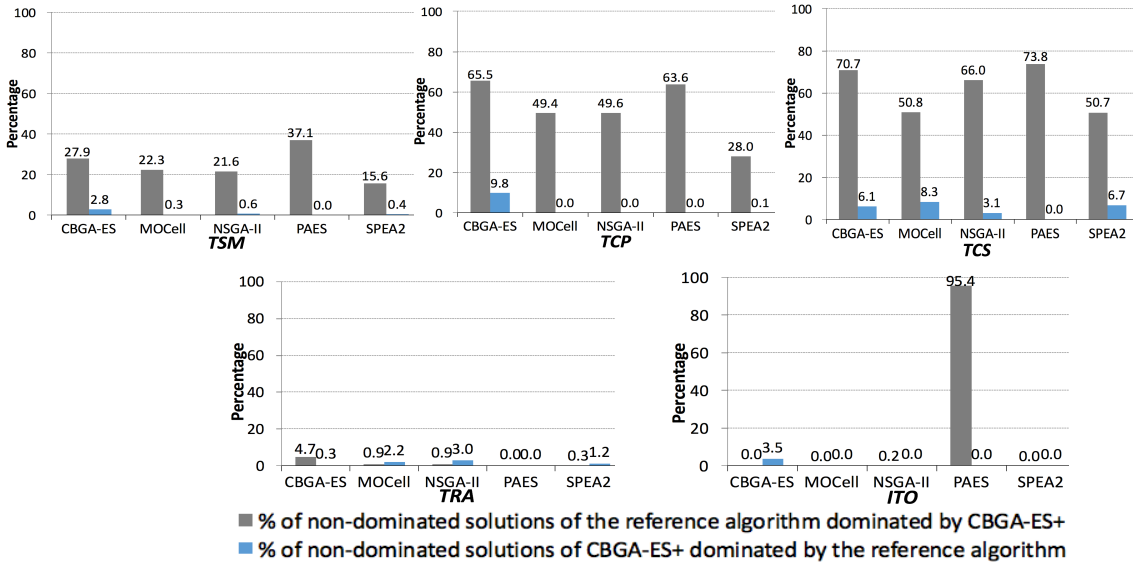

Fig. C-7. Percentage of non-dominated solutions by/of CBGA-ES$^+$ with reference to the search algorithms*

*Reference algorithm refers to CBGA-ES, MOCell, NSGA-II, PAES, or SPEA2, which is below the bar in the bar graphs.

Although the solutions produced by CBGA-ES$^+$ had worse overall quality solutions (as indicated by *HV*) for *TSM* and *TCP* as compared to the selected search algorithms, they had higher fault detection rates (as indicated by *FD* and *APFD*) for most of the experiments. This is because CBGA-ES$^+$ favors elite solutions and tend to produce better quality solutions. Therefore, even though the solutions generated by CBGA-ES$^+$ have less diversity (as indicated by *GS* in Section 6.1.2), they have a better convergence (as indicated by *GD*) and overall better quality solutions (as indicated by *HV* in Section 6.1.2) on average. For D_5, CBGA-ES$^+$ did not have higher fault detection rates than the selected algorithms even though it has higher values for the objectives because higher values for the objectives do not always imply better fault detection. For instance, the test case that failed in the last version of D_5 never failed before, and it did not have high statement coverage. Therefore, the failing test case had a lower value for the objectives.

8 Threats to Validity

Threats to *construct validity* exists when the comparison measurements are not comparable for all the search algorithms [74]. To reduce *construct validity* threats, we used the same stopping criteria (i.e., 50,000 fitness evaluations) to find optimal solutions for all five multi-objective test optimization problems across the eight case studies. Another threat to *construct validity* arises when the measurement metrics do not sufficiently cover the concepts they are supposed to measure [8]. To mitigate this threat, we compared the different algorithms using the widely used evaluation metrics (e.g., *HV*, *GS*, *GD*, *FD,* and *APFD*).

Threats to *internal validity* consider the internal factors (e.g., algorithm parameters) that could influence the obtained results [75]. In our experiments, we used the iRace optimization package [69] to run the algorithms with their best settings for the different case studies. Another possible internal validity concerns about the implementation of the algorithms. To mitigate this threat, we implemented all the experimented algorithms in the same tool (jMetal) and the genetic operators have used the same implementation settings.

Threats to *conclusion validity* relate to the factors that influence the conclusion drawn from the results of the experiments [76]. The *conclusion validity* threat when using randomized algorithms is related to random variation in the produced results. To mitigate this threat, we repeated each experiment 50 times for each algorithm to reduce the possibility that the results were obtained by chance. Moreover, following the guidelines of reporting results for randomized algorithms [59], we employed the Vargha and Delaney statistics test 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.

Threats to *external validity* are related to the factors that affect the generalization of the results [75]. The first threat to *external validity* is the selection of search algorithms for our experiments. To mitigate this threat, we selected four different search algorithms that have managed to get good results in the literature [2, 3, 5, 17, 25] and the earlier version of the algorithm, CBGA-ES. The second threat to *external validity* concerns the number of case studies used to verify the results. To mitigate this threat, we used eight different case studies (that includes two industrial, one real world, and five open source case studies) focusing on five multi-objective test optimization problems (i.e., *TSM, TCP, TCS, TRA,* and *ITO*) for evaluating CBGA-ES^{+}. Moreover, we have selected the five multi-objective test optimization problems from a well-known search based software engineering repository [39]. It is also worth mentioning that such threats to *external validity* are common in empirical studies [16, 64].

9 Related Work

Existing studies have shown that Search-Based Software Testing (SBST) can achieve promising results by employing multi-objective search algorithms (e.g., NSGA-II) for dealing with distinct testing problems [10, 12-14, 25, 36, 77-79]. A review of a set of testing problems where SBST has been applied is presented in [80, 81].

9.1 Multi-Objective Test Optimization Problems

There is a large body of research using search-based approaches to solve multi-objective test optimization problems [10-14, 36, 38, 77-79, 82-84]. For instance, Epitropakis et al. [82] defined three objectives (e.g., fault history coverage) for the test case prioritization problem and solved it using seven algorithms (e.g., NSGA-II). Yoo and Harman [12] defined three objectives (e.g., code coverage) for the test case selection problem and solved it using a greedy algorithm and NSGA-II. Wang et al. [38] defined four objectives (e.g., test resource usage) for test case prioritization and solved it using seven algorithms (e.g., SPEA2, PAES, NSGA-II). For the testing resource allocation under uncertainty problem, Pietrantuono et al. [83] defined three objectives (e.g., fault correction) and solved it using four algorithms (e.g., MOCELL, PAES).

Similarly, in our earlier work for addressing test suite minimization problem [2], we defined five objectives (e.g., fault detection capability) and empirically evaluated nine multi-objective search algorithms (e.g., NSGA-II). For prioritizing test cases, in our previous work [3], we defined four objectives (e.g., configuration coverage) and empirically evaluated NSGA-II with RS and Greedy. With respect to test case selection problem, four objectives (e.g., mean priority) were defined and in total seven multi-objective search algorithms (e.g., SPEA2, PAES) were empirically evaluated [17]. Regarding the testing resource allocation problem, Wang et al. [5] defined three objectives (e.g., reliability) and empirically evaluated two multi-objective search algorithms (e.g., NSGA-II). With respect to the integration and test order problem, Assunção et al. [25] defined four objectives (e.g., number of attributes) that was also investigated in [35, 40, 85], and empirically evaluated three multi-objective search algorithms (e.g., NSGA-II).

As compared with the previous studies for multi-objective test optimization problems mentioned above, this work has a different focus, i.e., proposing a new algorithm (i.e., CBGA-ES$^+$) by introducing non-dominated elitist strategy with the aim to further reduce the randomness that exists in the existing multi-objective search algorithms. Moreover, by employing the five case studies that were used in [2-5, 17, 25, 35] and three open source case studies from [44], we compared the performance of CBGA-ES$^+$ with its predecessor CBGA-ES and four selected search algorithms that managed to achieve good results in the existing works [2, 3, 5, 17, 25]. Results show that CBGA-ES$^+$ significantly outperformed 1) the four selected algorithms for solving all the five selected multi-objective test optimization problems and 2) CBGA-ES for solving four multi-objective test optimization

problems while there was no difference for one multi-objective test optimization problem (i.e., integration and test order problem).

9.2 Multi-Objective Search Algorithms

Pareto-efficient multi-objective evolutionary algorithms are being increasingly used for multi-objective test suite optimization to simultaneously optimize multiple objectives. There exist several multi-objective search algorithms that have been applied to address multi-objective test optimization problems [15, 16, 22-24, 67, 86]. For instance, Deb et al. [15] proposed NSGA-II, which ensures that non-dominated solutions are preferred over the dominated ones and if the number of selected solution exceeds the population size, the solutions with a higher value of *crowding distance* are selected (recall Section 2.2). SPEA2 [22] combines the raw fitness of each solution with the density information and uses the non-dominated solutions from the population to fill the archive (Section 2.2). In [16] an algorithm, User-Preference Multi-Objective Optimization Algorithm (UPMOA) is proposed to incorporate the user preference for the different objectives. UPMOA is a modification of NSGA-II where the *crowding distance* indicator is replaced with the user-preference indicator based on existing weight assignment strategies (e.g., fixed weights).

As compared with the existing multi-objective algorithms, CBGA-ES$^+$ is a new multi-objective search algorithm that divides the population into different clusters and ranks the clusters based on the defined *cluster dominance strategy* (Section 3.1). The non-dominated solutions are further selected from the best clusters until the size of the selected population reaches the defined elite population minimum size.

In our conference paper, we proposed CBGA-ES [21] that 1) divides the initial population into different clusters, 2) uses the defined *cluster dominance strategy* to rank the different clusters, and 3) selects the solutions from the best cluster (i.e., elite solutions) for producing offspring solutions. However, as compared to CBGA-ES, CBGA-ES$^+$ further introduces non-dominated elitist selection strategy by selecting only the non-dominated elite solutions for producing the offspring solutions (Section 3.2).

9.3 Cluster-Based Search Algorithms

Multi-objective GA together with clustering has been increasingly applied for multi-objective optimization of real-life applications such as software module clustering [87], web mining [88], and time series data analysis [89]. For instance, Aibinu et al. [90] proposed a clustering based GA (CGA) with the polygamy selection and dynamic population control for route optimization. CGA initializes candidate solutions based on the problem definition and clusters the solutions into two non-overlapping clusters. Solutions from the cluster with better fitness values are used for producing offspring using the *crossover* and *mutation* operators.

Li et al. [91] proposed a clustering based GA for addressing four multi-objective benchmark problems from the literature [92]. The proposed GA [91] divided the whole

population into n clusters and placed similar solutions in a cluster. Afterward, the non-dominated solutions from each cluster were gathered using the Arena principle [93], where each solution was compared with every other solution in the cluster to see if it was dominated. Zhu et al. [94] proposed a cluster-based orthogonal multi-objective genetic algorithm and evaluated it using five multi-objective benchmark problems from [15]. An initial population is generated by the orthogonal design, which scatters the individual solutions evenly in the feasible solution space [95]. The initial population was then partitioned into several clusters, and new solutions were generated in each cluster by using the *crossover* operator. Non-dominated sorting and crowding distance sorting [15] were further used to create a new population by selecting elite solutions from the parent population and generate offspring in each cluster.

As compared with the above-mentioned existing studies, CBGA-ES$^+$ is different from at least two perspectives: 1) we defined the *cluster dominance strategy* (Section 3.1) to determine the quality of different clusters; and the non-dominated solutions from the best clusters can be chosen for generating offspring solutions; 2) we defined the elite population minimum size to ensure that there are enough solutions in the population for maintaining diversity, which is not the case in the existing works.

Moreover, niching approaches have been used in genetic algorithms to form niches (i.e., groups) of individuals (i.e., solutions) in a population [96]. The individuals within a niche are similar to each other and different across niches. Fitness sharing [97] is the most well-known niching approach, which uses the sharing radius (σ_s) to group together similar individuals. If the distance between the two individuals is lower than σ_s, they are placed in the same niche [96]. The distance between two individuals can be measured by using different distance metrics, e.g., Hamming distance and Euclidean distance. The individuals within the same niche share the fitness values that can be measured by $f_i' = \frac{f_i}{m_i}$, where f_i is the original fitness value of an individual i and m_i refers to the number of individuals in the niche [96]. Notice that the shared fitness values of individuals decrease when new individuals are introduced into the niche [96].

Our work (i.e., CBGA-ES$^+$) is different from [96, 97] from at least two perspectives: 1) fitness sharing dynamically changes the shared fitness values for individuals whereas CBGA-ES$^+$ does not influence the fitness values of individuals during the search process; and 2) fitness sharing uses the shared fitness values to compare individuals for obtaining optimal solutions while CBGA-ES$^+$ employs the defined *cluster dominance strategy* (Section 3.1) to compare clusters for producing optimal solutions.

10 Conclusion and Future Work

This paper proposed a cluster-based genetic algorithm with non-dominated elitist selection (CBGA-ES$^+$) for addressing multi-objective test optimization problems. CBGA-ES$^+$ was empirically evaluated by comparing with its predecessor CBGA-ES (Section 3.2) and four

search algorithms (i.e., MOCELL, NSGA-II, PAES, and SPEA2), which performed well to address various multi-objective test optimization problems [2, 3, 5, 17, 25]. Results from the eight subjects show that CBGA-ES$^+$ significantly outperformed its predecessor CBGA-ES and the four selected algorithms for most of the considered multi-objective test optimization problems: test suite minimization (*TSM*), test case prioritization (*TCP*), test case selection (*TCS*), testing resource allocation (*TRA*), and integration and test order (*ITO*) problem.

Shortly, we plan to apply more multi-objective software engineering optimization problems from different domains (e.g., requirement assignment problems [98]) to further strengthen CBGA-ES$^+$. We also want to involve industrial practitioners to deploy and assess CBGA-ES$^+$ in real industrial settings and study the impact of fitness evaluations (i.e., termination criteria) on the performance of CBGA-ES$^+$.

Acknowledgement

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

[1] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pp. 1493-1500. ACM, 2013.

[2] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software,* vol. 103, pp. 370-391. Elsevier, 2015.

[3] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "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*, pp. 172-190. Springer, 2016.

[4] Y.-S. Dai, M. Xie, K.-L. Poh, and B. Yang, "Optimal testing-resource allocation with genetic algorithm for modular software systems," *Journal of Systems and Software,* vol. 66, no. 1, pp. 47-55. Elsevier, 2003.

[5] Z. Wang, K. Tang, and X. Yao, "A multi-objective approach to testing resource allocation in modular software systems," in *IEEE Congress on Evolutionary Computation*, pp. 1148-1153. IEEE, 2008.

[6] O. Bühler and J. Wegener, "Evolutionary functional testing," *Computers & Operations Research,* vol. 35, no. 10, pp. 3144-3160. Elsevier, 2008.

[7] L. C. Briand, J. Feng, and Y. Labiche, "Using genetic algorithms and coupling measures to devise optimal integration test orders," in *Proceedings of the 14th*

International Conference on Software Engineering and Knowledge Engineering, pp. 43-50. ACM, 2002.

[8] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering,* vol. 33, no. 4, pp. 225-237. IEEE, 2007.

[9] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, pp. 1021-1028. ACM, 2005.

[10] W. Sun, Z. Gao, W. Yang, C. Fang, and Z. Chen, "Multi-objective test case prioritization for GUI applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1074-1079. ACM, 2013.

[11] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software,* vol. 83, no. 4, pp. 689-701. Elsevier, 2010.

[12] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 140-150. ACM, 2007.

[13] S. Yoo, M. Harman, and S. Ur, "Highly scalable multi objective test suite minimisation using graphics cards," in *International Symposium on Search Based Software Engineering*, pp. 219-236. Springer, 2011.

[14] Z. Li, Y. Bian, R. Zhao, and J. Cheng, "A fine-grained parallel multi-objective test case prioritization on GPU," in *Proceedings of the International Symposium on Search Based Software Engineering*, pp. 111-125. Springer, 2013.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* vol. 6, no. 2, pp. 182-197. IEEE, 2002.

[16] S. Wang, S. Ali, T. Yue, and M. Liaaen, "UPMOA: An improved search algorithm to support user-preference multi-objective optimization," in *Proceedings of the 26th International Symposium on Software Reliability Engineering*, pp. 393-404. IEEE, 2015.

[17] D. Pradhan, S. Wang, S. Ali, and T. Yue, "Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1085-1092. ACM, 2016.

[18] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," *Foundations of Genetic Algorithms,* vol. 1, pp. 69-93. Morgan Kaufmann Publishers, 1991.

[19] A. Shukla, H. M. Pandey, and D. Mehrotra, "Comparative review of selection techniques in genetic algorithm," in *International Conference on Futuristic Trends on Computational Analysis and Knowledge Management*, pp. 515-519. IEEE, 2015.

[20] A. Brindle, "Genetic algorithms for function optimization," Doctoral dissertaion and Technical Report TR81-2, University of Alberta, Department of Computer Science, Edmonton, 1981.

[21] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "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*, pp. 367-378. IEEE, 2017.

[22] E. Zitzler, M. Laumanns, and L. Thiele, "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), 2001.

[23] J. Knowles and D. Corne, "The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation," in *Proceedings of the Congress on Evolutionary Computation*. IEEE, 1999.

[24] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "Mocell: A cellular genetic algorithm for multiobjective optimization," *International Journal of Intelligent Systems,* vol. 24, no. 7, pp. 726-746. Wiley Online Library, 2009.

[25] W. K. G. Assunção, T. E. Colanzi, S. R. Vergilio, and A. Pozo, "A multi-objective optimization approach for the integration and test order problem," *Information Sciences,* vol. 267, pp. 119-139. Elsevier, 2014.

[26] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary Computation,* vol. 8, no. 2, pp. 173-195. MIT Press, 2000.

[27] A. S. Sayyad and H. Ammar, "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*, pp. 21-27. IEEE, 2013.

[28] N. Srinvas and K. Deb, "Multi-objective function optimization using non-dominated sorting genetic algorithms," *Evolutionary Computation,* vol. 2, no. 3, pp. 221-248. MIT Press, 1994.

[29] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "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*, pp. 631-642. ACM, 2016.

[30] C. R. Reeves, *Modern heuristic techniques for combinatorial problems*: John Wiley & Sons, Inc., 1993.

[31] J. Brownlee, *Clever algorithms: nature-inspired programming recipes*: Lulu, 2011.

[32] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety,* vol. 91, no. 9, pp. 992-1007. Elsevier, 2006.

[33] S. Li, N. Bian, Z. Chen, D. You, and Y. He, "A simulation study on some search algorithms for regression test case prioritization," in *International Conference on Quality Software (QSIC)*, pp. 72-81. IEEE, 2010.

[34] P. Hajela and C.-Y. Lin, "Genetic search strategies in multicriterion optimal design," *Structural optimization,* vol. 4, no. 2, pp. 99-107. 1992.

[35] G. Guizzo, S. R. Vergilio, A. T. Pozo, and G. M. Fritsche, "A multi-objective and evolutionary hyper-heuristic applied to the Integration and Test Order Problem," *Applied Soft Computing,* vol. 56, pp. 331-344. Elsevier, 2017.

[36] Z. Wang, K. Tang, and X. Yao, "Multi-objective approaches to optimal testing resource allocation in modular software systems," *IEEE Transactions on Reliability,* vol. 59, no. 3, pp. 563-575. IEEE, 2010.

[37] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability,* vol. 22, no. 2, pp. 67-120. Wiley Online Library, 2012.

[38] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "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*, pp. 182-191. ACM, 2016.

[39] Y. Zhang, M. Harman, and A. Mansouri, "The SBSE repository: A repository and analysis of authors and research articles on search based software engineering," *crestweb. cs. ucl. ac. uk/resources/sbse repository,* 2012.

[40] W. K. G. Assunção, T. E. Colanzi, A. T. R. Pozo, and S. R. Vergilio, "Establishing integration test orders of classes with several coupling measures," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pp. 1867-1874. ACM, 2011.

[41] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory,* vol. 28, no. 2, pp. 129-137. IEEE, 1982.

[42] Y. Liang and K.-S. Leung, "Genetic Algorithm with adaptive elitist-population strategies for multimodal function optimization," *Applied Soft Computing,* vol. 11, no. 2, pp. 2017-2034. Elsevier, 2011.

[43] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281-297. University of California Press, 1967.

[44] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 437-440. ACM, 2014.

[45] "ISO 13628-6:2006," in *Design and operation of subsea production systems -- Part 6: Subsea production control systems*, ed, 2006.

[46] *OREDA Offshore Reliability Data Handbook 2002, 4th edition* Høvik, Norway : OREDA Participants : Distributed by Der Norske Veritas, 2002.

[47] *Class TestCase*. Available:

http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html

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

[49] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering,* vol. 42, no. 5, pp. 490-505. 2016.

[50] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach," *IEEE Transactions on Evolutionary Computation,* vol. 3, no. 4, pp. 257-271. IEEE, 1999.

[51] J. Knowles, L. Thiele, and E. Zitzler, "A tutorial on the performance assessment of stochastic multiobjective optimizers," *Tik report,* vol. 214, pp. 327-332. ETH Zurich, 2006.

[52] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham, "AbYSS: Adapting scatter search to multiobjective optimization," *IEEE Transactions on Evolutionary Computation,* vol. 12, no. 4, pp. 439-457. IEEE, 2008.

[53] A. Zhou, Y. Jin, Q. Zhang, B. Sendhoff, and E. Tsang, "Combining model-based and genetics-based offspring generation for multi-objective optimization using a convergence criterion," in *IEEE Congress on Evolutionary Computation (CEC), 2006.*, pp. 892-899. IEEE, 2006.

[54] D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithm research: A history and analysis," Technical Report TR-98-03, Department of Electrical and Computer Engineering, Graduate School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio1998.

[55] L. Vidács, Á. Beszédes, D. Tengeri, I. Siket, and T. Gyimóthy, "Test suite reduction for fault detection and localization: A combined approach," in *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014*, pp. 204-213. IEEE, 2014.

[56] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on software engineering,* vol. 22, no. 8, pp. 529-551. 1996.

[57] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 179-188. IEEE, 1999.

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

[59] A. Arcuri and L. Briand, "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)*, pp. 1-10. IEEE, 2011.

[60] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132. 2000.

[61] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics,* pp. 50-60. JSTOR, 1947.

[62] G. Neumann, M. Harman, and S. Poulding, "Transformed vargha-delaney effect size," in *Proceedings of the International Symposium on Search Based Software Engineering*, pp. 318-324. Springer, 2015.

[63] W. Martin, F. Sarro, and M. Harman, "Causal impact analysis for app releases in google play," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering*, pp. 435-446. ACM, 2016.

[64] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 619-630. ACM, 2016.

[65] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software,* vol. 42, no. 10, pp. 760-771. Elsevier, 2011.

[66] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba, "Solving three-objective optimization problems using a new hybrid cellular genetic algorithm," in *Parallel Problem Solving from Nature–PPSN X*, ed: Springer, 2008, pp. 661-670.

[67] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. C. Coello, F. Luna, and E. Alba, "Smpso: A new pso-based metaheuristic for multi-objective optimization," in *Proceedings of the Symposium on Computational Intelligence in Multi-criteria Decision-making*, pp. 66-73. IEEE, 2009.

[68] A. S. Sayyad, T. Menzies, and H. Ammar, "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*, pp. 492-501. IEEE, 2013.

[69] M. López-Ibánez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package, iterated race for automatic algorithm configuration," Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium2011.

[70] M. R. Hoffmann, J. Brock, and E. Mandrikov. *Java Code Coverage for Eclipse*. Available: http://www.eclemma.org/

[71] *The Abel computer cluster*. Available: http://www.uio.no/english/services/it/research/hpc/abel/

[72] A. M. Pitangueira, P. Tonella, A. Susi, R. S. P. Maciel, and M. Barros, "Minimizing the stakeholder dissatisfaction risk in requirement selection for next release planning," *Information and Software Technology,* vol. 87, pp. 104-118. 2017.

[73] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Software Engineering Notes,* vol. 31, no. 1, pp. 35-42. ACM, 2006.

[74] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Software,* vol. 12, no. 4, p. 52. IEEE, 1995.

[75] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*: John Wiley & Sons, 2012.

[76] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, "Experimentation in software engineering: an introduction. 2000," ed: Kluwer Academic Publishers, 2000.

[77] Q. Gu, B. Tang, and D. Chen, "Optimal regression testing based on selective coverage of test requirements," in *International Symposium on Parallel and Distributed Processing with Applications*, pp. 419-426. IEEE, 2010.

[78] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 329-338. IEEE Computer Society, 2001.

[79] C. L. B. Maia, R. A. F. do Carmo, F. G. de Freitas, G. A. L. de Campos, and J. T. de Souza, "A multi-objective approach for the regression test case selection problem," in *Proceedings of the Anais do XLI Simposio Brasileiro de Pesquisa Operacional*, pp. 1824-1835. SBPO, 2009.

[80] P. McMinn, "Search-based software testing: Past, present and future," in *IEEE Fourth International Conference onSoftware Testing, Verification and Validation Workshops*, pp. 153-163. IEEE, 2011.

[81] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *IEEE International Conference on Software Testing, Verification and Validation*, pp. 1-12. IEEE, 2015.

[82] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 234-245. ACM, 2015.

[83] R. Pietrantuono, P. Potena, A. Pecchia, D. Rodriguez, S. Russo, and L. Fernandez, "Multi-Objective Testing Resource Allocation under Uncertainty," *IEEE Transactions on Evolutionary Computation,* IEEE, 2017.

[84] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 1-12. ACM, 2006.

[85] T. Colanzi, W. Assunção, S. Vergilio, and A. Pozo, "Integration test of classes and aspects with a multi-evolutionary and coupling-based approach," *Search Based Software Engineering,* pp. 188-203. Springer, 2011.

[86] L. Thiele, K. Miettinen, P. J. Korhonen, and J. Molina, "A preference-based evolutionary algorithm for multi-objective optimization," *Evolutionary Computation,* vol. 17, no. 3, pp. 411-436. MIT Press, 2009.

[87] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering,* vol. 37, no. 2, pp. 264-282. IEEE, 2011.

[88] G. N. Demir, A. S. Uyar, and S. G. Ögüdücü, "Graph-based sequence clustering through multiobjective evolutionary algorithms for web recommender systems," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pp. 1943-1950. ACM, 2007.

[89] S. Bandyopadhyay, U. Maulik, and R. Baragona, "Clustering multivariate time series by genetic multiobjective optimization," *Metron- International Journal of Statistics,* vol. 68, no. 2, pp. 161-183. Springer, 2010.

[90] A. Aibinu, H. B. Salau, N. A. Rahman, M. Nwohu, and C. Akachukwu, "A novel Clustering based Genetic Algorithm for route optimization," *Engineering Science and Technology, an International Journal,* pp. 2022-2034. Elsevier, 2016.

[91] W. Li, G. Guo, and T. Yan, "A Multi-objective Genetic Algorithm Based on Clustering," in *Proceedings of the Second International Conference on Intelligent System Design and Engineering Application*, pp. 41-43. IEEE, 2012.

[92] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable multi-objective optimization test problems," in *Proceedings of the Congress on Evolutionary Computation*, pp. 825-830. IEEE, 2002.

[93] J.-H. Zheng, H. Jiang, D. Kuang, and Z.-Z. Shi, "Approach of constructing multi-objective Pareto optimal solutions using arena's principle," *Ruan Jian Xue Bao (Journal of Software),* vol. 18, no. 6, pp. 1287-1297. Chinese Academy of Science, 2007.

[94] J. Zhu, G. Dai, and L. Mo, "A Cluster-Based Orthogonal Multi-Objective Genetic Algorithm," in *International Symposium on Intelligence Computation and Applications*, pp. 45-55. Springer, 2009.

[95] D. C. Montgomery, *Design and analysis of experiments*. New York: John Wiley & Sons, 1991.

[96] B. Sareni and L. Krahenbuhl, "Fitness sharing and niching methods revisited," *IEEE Transactions on Evolutionary computation,* vol. 2, no. 3, pp. 97-106. IEEE, 1998.

[97] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," in *Proceedings of the Second International Conference on Genetic Algorithms: Genetic algorithms and their applications*, pp. 41-49. L. Erlbaum Associates Inc., 1987.

[98] Y. Li, T. Yue, S. Ali, and L. Zhang, "Zen-ReqOptimizer: a search-based approach for requirements assignment optimization," *Empirical Software Engineering,* pp. 1-60. Springer, 2016.

Paper D

Search-Based Test Case Implantation for Testing Untested Configurations

Dipesh Pradhan, Shuai Wang, Tao Yue,
Shaukat Ali, Marius Liaaen

Revision submitted to Journal of Information and Software Technology (IST), September 2018.

Abstract

***Context**: Modern large-scale software systems are highly configurable, and thus require a large number of test cases to be implemented and revised for testing a variety of system configurations. This makes testing highly configurable systems very expensive and time-consuming.*

***Objective**: Driven by our industrial collaboration with a video conferencing company, we aim to automatically analyze and implant existing test cases (i.e., an original test suite) to test the untested configurations.*

***Method**: We propose a search-based test case implantation approach (named as SBI) consisting of two key components: 1) Test case analyzer that statically analyzes each test case in the original test suite to obtain the program dependence graph for test case statements and 2) Test case implanter that uses multi-objective search to select suitable test cases for implantation using three operators, i.e., selection, crossover, and mutation (at the test suite level) and implants the selected test cases using a mutation operator at the test case level including three operations (i.e., addition, modification, and deletion).*

***Results**: We empirically evaluated SBI with an industrial case study and an open source case study by comparing the implanted test suites produced by three variants of SBI with the original test suite using evaluation metrics such as statement coverage (SC), branch coverage (BC), and mutation score (MS). Results show that for both the case studies, the test suites implanted by the three variants of SBI performed significantly better than the original test suites. The best variant of SBI achieved on average 19.3% higher coverage of configuration variable values for both the case studies. Moreover, for the open source case study, the best variant of SBI managed to improve SC, BC, and MS with 5.0%, 7.9%, and 3.2%, respectively.*

***Conclusion**: SBI can be applied to automatically implant a test suite with the aim of testing untested configurations and thus achieving higher configuration coverage.*

Keywords: Search; Multi-Objective Optimization; Genetic Algorithms; Test Case Implantation.

1 Introduction

Testing plays a key role to ensure that software systems can be released to market with high quality and more than 50% of time and budget are spent for testing [1]. It is even significantly worse when testing large-scale software systems that are usually highly configurable since test engineers need to spend a great deal of effort to implement and revise test cases for testing various configurations, which decreases the efficiency of testing [2].

We have been working with a video conferencing company since 2009 with the aim to assist their current practice of testing large-scale Video Conferencing Systems (VCSs). For each VCS, there are more than 100 configuration variables (e.g., *protocol*), and each variable can be configured with a number of values (e.g., *protocol* can be *SIP* and *H323*). Such highly configurable VCSs bring great challenges for test engineers to manually and systematically design and develop test cases. For example, the *calltype* indicating a particular call type can be configured as *video* or *audio* and the *callrate* that specifies the call rate to be used when placing or receiving video calls can be configured with an integer from 64 to 6000. For the VCSs that support these two configuration variables, there are in total 2×5,937 =11,874 configurations that are needed to be thoroughly tested. For each configuration, a set of test API commands with a number of parameters has to be called (e.g., dial (*calltype=video*, *callrate= 64*)) and a set of corresponding system status variables need to be checked (e.g., assert (*activecalls*=1,*videocalls*=1)).

Manually implementing such test cases (e.g., specifying configurations, calling relevant test API commands, checking corresponding system status) to test configurations require a large amount of manual work, which is practically infeasible. Test engineers at the company usually choose to develop a certain number of test cases by including a limited number of configurations (e.g., *video* with *callrate 6000*) based on their experience. Such practice may result in high chances that potential errors cannot be detected since some configurations might not be covered during testing. Our industrial partner develops VCSs in a continuous integration environment and 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. The median execution time of a test case is 30 minutes, and thus, they need the existing test cases to be as efficient as possible, i.e., testing different configurations to increase the configuration coverage.

With the above-mentioned challenges in mind, we believe that it is worth investigating how to automatically and systematically analyze and implant existing test cases to increase the overall configuration coverage and thereby improve the efficiency of testing. Therefore, we propose a search-based test case implantation approach (named as SBI) to automatically analyze and implant an existing test suite with the aim to test the untested configurations. More specifically, SBI includes two key components: 1) *Test case analyzer* that statically analyzes each test case in the original test suite to obtain the program dependence graph for the statements; and 2) *Test case implanter* that uses multi-objective search to select suitable test cases for implantation using three *operators*, i.e., *select*ion, *crossover*, and *mutation* (at the test suite level) and implants the selected test cases using a *mutation operator* at the test case level that includes three operations: *addition*, *modification,* and *deletion*. To assess the quality of the implanted test suites, we define five cost-effectiveness measures: number of configuration variable values covered ($NCVV$), pairwise coverage of parameter values of test API commands ($PCPV$), number of implanted test cases (NIT), number of changed statements (NCS), and estimated execution time (EET).

We evaluated three variants of SBI (using Non-dominated Sorting Genetic Algorithm II (NSGA-II) [3], weight-based genetic algorithm (WGA), and random search (RS)) with one industrial case study from a video conferencing company with a test suite including 118 test cases and one open-source case study (i.e., SafeHome [4]) with 94 test cases. We also applied three evaluation metrics: statement coverage (SC), branch coverage (BC), and mutation score (MS) to evaluate the three variants of SBI for the SafeHome case study by generating in total 1594 non-equivalent mutants. Note that we cannot apply these metrics (i.e., SC, BC, and MS) to the industrial case study since we do not have access to the source code. The evaluation results showed that the implanted test suites produced by all the three variants of SBI significantly outperformed the original suite for both the case studies. Among the different SBI variants, SBI with NSGA-II (i.e., $SBI_{NSGA\text{-}II}$) performed the best. Specifically, it achieved on average 19.3% higher $NCVV$ and 57.0% higher $PCPV$. Moreover, for the SafeHome case study, the test suites implanted by $SBI_{NSGA\text{-}II}$ managed to improve SC, BC and MS with on average 5.0%, 7.9%, and 3.2%, respectively.

The key contributions of this paper include:

1) A formalization of the test case implantation problem (Section 4.2);
2) A mathematical definition of the five cost-effectiveness measures to assess the quality of implanted test suites (Section 4.3);
3) SBI: A novel search-based test case implantation approach with two key components, i.e., *test case analyzer* and *test case implanter* (Section 5);
4) An empirical evaluation of the three SBI variants (with NSGA-II, WGA, and RS) using the two case studies (Section 7).

The rest of the paper is organized as follows: Section 2 gives relevant background. Section 3 introduces a running example for illustrating SBI and the overall context, followed by the formalization of the problem (Section 4). Section 5 presents SBI in detail, followed by the experiment design (Section 6) and results of the empirical study (Section 7). Section 8 presents the threats to validity. Section 9 discusses the related work, and Section 10 concludes the paper.

2 Background

2.1 Multi-Objective Test Optimization

Multi-objective test optimization aims to find solutions for software testing problems with tradeoff relationships among objectives (e.g., maximizing code coverage while minimizing execution cost), such as test case prioritization (*TCP*) [5, 6]. With multi-objective test optimization, a set of solutions with equivalent quality is usually produced based on *Pareto optimality* if there exists more than one best solution [7]. More specifically, *Pareto optimality* defines the Pareto dominance to assess the quality of solutions. Consider that there are a objectives $= \{o_1, o_2, \dots, o_a\}$ to be optimized for a multi-objective test optimization problem (e.g., *TCP*), and each objective can be calculated using a fitness

function f_i from $F = \{f_1, f_2, \ldots, f_a\}$. Then if we aim to minimize the fitness function such that a lower value for an objective implies better performance, then solution Y dominates solution Z (i.e., $Y \succ Z$) *iff* $\forall_{i=1,2,\ldots,a} f_i(Y) \leq f_i(Z) \wedge \exists_{i=1,2,\ldots,a} f_i(Y) < f_i(Z)$. The test optimization approaches specific to this work are discussed in the related work section (Section 9).

2.2 Genetic Algorithms

Genetic Algorithms (GAs) are inspired by the process of natural selection to optimize one or more objectives (e.g., maximizing code coverage while minimizing the execution cost) using a fitness function(s) to assess the quality of solutions. A typical GA uses bio-inspired operators (i.e., *selection*, *crossover*, and *mutation*) to produce offspring solutions [8]. The *selection* operator selects the best solutions based on the fitness functions, the *crossover* operator partially exchanges the parent solutions, and the *mutation* operator mutates a given solution by changing part of the solutions.

Weight-based GA (WGA) assigns a particular weight to each objective function (e.g., each objective is assigned equal weight if all the objectives hold equal importance) for converting a multi-objective problem into a single objective problem using a scalar objective function [9]. Non-dominated Sorting Genetic Algorithm II (NSGA-II) is a computationally fast and elitist multi-objective evolutionary algorithm [3]. In NSGA-II, solutions in a population are sorted and placed into several fronts based on the ordering of Pareto dominance. Individual solutions are selected from non-dominated fronts, and if the number of solutions from a non-dominated front exceeds the specified population size, the solution with a higher value of *crowding distance* is selected, where *crowding distance* is used to measure the distance between individual solutions in a population [9].

3 Running Example and Context

The running example is an excerpt of a sanitized test case from a video conferencing company, which will be used to illustrate SBI throughout the paper. A typical test case at the video conferencing company consists of one *setup class,* one or more *test methods,* one *teardown,* and one *teardown class* (Table D-1) as recommended in the unit testing framework in python, PyUnit [10]. A *setup class* is for initializing and setting up the system under test (SUT) (e.g., registering SUT to a registrar at line 1 in Table D-1) to be ready for executing the *test methods* in the test case. The *test methods* are for testing SUT functionalities (e.g., the *dial* functionality for making a call from one system to another, as shown at line 4 in Table D-1). *Teardown* resets the SUT (e.g., *disconnect* the SUT, as shown at line 7 in Table D-1), and it is executed after each *test method* has been called. Lastly, *teardown class* is called after all the *test methods* have been executed to reset the statuses of the SUT that might have been modified at the *setup class* (e.g., disconnecting the SUT from the registrar).

Moreover, Fig. D-1 presents an overview of a typical testing process for testing VCSs, i.e., SUTs. As a first step, a test case makes the SUT ready for testing, e.g., registering the SUT to a registrar (line 1 in Table D-1) as a part of *setup class*. Secondly, the SUT is configured if necessary. For example, the configuration variable *packetlossresilence* at the SUT is configured with *off* in the *test method* (line 2 in Table D-1). In the third step, one or more test API command is executed on the SUT. For example, *dial* is executed, which consists of four parameters: *protocol*, *calltype, callrate,* and *autoanswer* assigned with values *sip*, *video, 6000,* and *true,* respectively in Table D-1. Then, the statuses of the SUT are verified with an *assertion*. For example, the *assertion* checks if the values of *NumberOfActiveCalls* and *NumberOfVideoCalls* are both 1 in Table D-1. Note that typically each *test method* consists of at least one test API command (e.g., *dial*) and an *assertion*. At last, the statuses of the SUT are reset to the original statuses, e.g., *disconnect* as a part of *teardown*. If the test case has more than one *test method*, the next *test method* is executed followed by the *teardown*, and this process is repeated until all the *test methods* in the test case are executed. At the final step, *teardown class* is called to reset the statuses of the SUT that might have been initialized at the *setup class*.

Table D-1. An Excerpt of a Sanitized Test Case

Part	Line	Example	Comment
Setup class	1	register SUT to a registrar	Register SUT
Test method	2	packetlossresilence = off	Configure SUT
	3	callrate_var = 6000	Assign variable
	4	dial(protocol=sip, calltype=video, callrate=callrate_var, autoanswer=true)	Execute test API command on SUT
	5	wait(4)	Wait 4 seconds
	6	assert(NumberOfActiveCalls=1, NumberofVideoCalls=1)	Verify statuses of SUT
Teardown	7	disconnect call	Reset statuses
Teardown class	8	disconnect SUT from the registrar	Execution completed

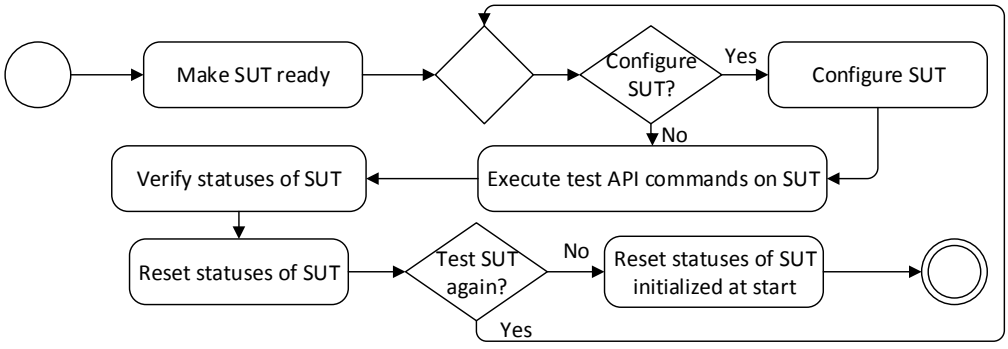

Fig. D-1. An overview of testing a VCS (SUT)

Each VCS developed by the video conferencing company is highly configurable. For example, a VCS includes more than 100 configuration variables (e.g., *packetlossresilence*) and each configuration variable can take a set of values (e.g., *packetlossresilence* can take

two values: *off* and *on*). Moreover, each test API command requires configuring one or more parameters (e.g., four parameters need to be configured for the test API command *dial*), and each parameter in the test API command can take a number of different values. For example, the test API command *dial* allows values *audio* and *video* for *calltype*, values between *64* and *6000* inclusive for the *callrate*, and values *true* and *false* for *autoanswer*.

Testing all the values for the configuration variables requires a large number of *test methods* if each *test method* covers one configuration variable. Moreover, testing all the combinations of parameter values for the test API commands also requires a large number of *test methods*. Additionally, if one *test method* includes more than one test API command, the combinations could exponentially increase, which makes the manual test case development expensive and even infeasible at certain contexts. Developing new test cases is practically expensive since each test case should include the *setup class, teardown,* and *teardown class*, which cause an extra overhead in terms of test case execution. This is due to the fact that the *setup class* (for setting up the SUT) and *teardown class* (for resetting the SUT) need to be executed for all the newly implemented test cases, which is quite time-consuming. However, if the new *test methods* can be directly added to the existing test cases, the overhead for executing the *setup class* and *teardown class* can be reduced and thereby improving the efficiency of testing. Moreover, using test reduction strategies (e.g., boundary value analysis [11-13]) can further reduce the number of combinations of variables/parameters without significantly decreasing the effectiveness of the test suite.

Additionally, some existing test cases might test the same combinations of values for the configuration variables and test API command parameters. For example, when different test engineers develop test cases that require *dial*, it is possible that the same values for the parameters *protocol*, *calltype, callrate,* and *autoanswer* are taken, which can decrease the efficiency of testing since a different combination of configuration variable or test API command parameters could have been used. Notably, more diverse test cases (e.g., in terms of combinations of parameter values in our context) can lead to higher efficiency of testing [14, 15].

Thus, the key objective of this work is to propose a cost-effective search-based approach to automatically implant an existing test suite with the aim to 1) achieve a higher coverage of configuration variable values, 2) cover more combinations of parameter values of test API commands, and 3) increase the efficiency of testing by modifying or removing redundant *test methods* that cover same configuration variable values or the same combinations of parameter values of test API commands.

4 Problem Representation and Measures

This section first defines basic notations (Section 4.1) and the test case implantation problem (Section 4.2), followed by presenting the five cost/effectiveness measures (Section 4.3).

4.1 Basic Notations

We assume that a given original test suite consists of n test cases $T = \{t_1, t_2, \dots, t_n\}$. Each test case is composed of four parts (as mentioned in Table D-1): *setup class*, o number of *test methods*, *teardown*, and *teardown class*, i.e., $t = sc \cup \{tm_1, \dots, tm_o\} \cup td \cup tdc$, where sc, tm_i, td and tdc represent *setup class*, *test method* i, *teardown*, and *teardown class*, respectively.

Table D-2. Different Types of Statements in a Test Method

Name	Description	Example
Assignment	Assign values to Numeric, Boolean, String variables	callrate_var = 6000
Conditional	If-then statement represented $p \rightarrow q$, where p is a hypothesis and q is a conclusion	if (wait > 4) accept
Configuration	Configure the SUT	*packetlossresilence* = off
Execution	Perform actions by executing test API commands on the SUT	dial (*protocol*=SIP, *calltype*=video, *callrate*=6000, *autoanswer*=true)
Assertion	Check the statuses of the SUT	assert(NumberofActiveCalls=1)
Wait	Hold the execution of the next statement(s) for a specific time	wait (4)

Each *test method* tm_i is composed of a sequence of i, q statements, $tm_i = \{st_{i,1}, \dots, st_{i,q}\}$ (e.g., the *test method* in Table D-1 has 5 statements). Thus, the total statements in a test case is: $ST = \uplus_{tm \in t} F(tm) \uplus F(sc) \uplus F(td) \uplus F(tdc)$, where $F(tm), F(sc), F(td)$ and $F(tdc)$ are functions that return all the statements in the *test method* tm, *setup class* sc, *teardown* td, and *teardown class* tdc, respectively. Moreover, ST is a multiset, which is a collection of objects (e.g., statements in this context) that allows the objects to occur more than once in a set [16]. To enable the implantation, we need to get the statements structured, and therefore we classify them into six categories as shown in Table D-2.

Each test case covers one or more configuration variables, and each configuration variable is configured with a configuration value. For example, in the running example (), the test case covers configuration variable *packetlossresilence*, which is configured by using the value *off*. We represent a set of r configuration variables for test suite T as $CV_T = \{cv_1, \dots, cv_r\}$. Thus, the configuration variable values covered by the test suite T are defined as:

$$CVV_T = \bigcup_{cv \in CV_T} F(cv) \qquad (1)$$

Each test case executes one or more test API commands, each of which has one or more parameters. Each parameter is configured with a specific value at a *test method* for a test case. For example, in Table D-1, the test case covers the test API command *dial*, which has four parameters: *protocol*, *calltype*, *callrate*, and *autoanswer* with the values of *SIP*, *Video*, *6000*, and *true*, respectively. We represent the u number of test API commands covered by the test suite T as $AC_T = \{ac_1, ac_2, \dots, ac_u\}$. Each test API command ac_i has i, v parameters (i.e., $ac_i = \{ap_{i,0}, \dots, ap_{i,v}\}$). Systematically considering interactions of parameters during testing can lead to a high chance of finding software faults [14, 15].

Moreover, exhaustive testing (i.e., testing all combinations of parameters) is very expensive in practice. Therefore, pairwise testing has been proposed to reduce the number of interactions of test API parameters meanwhile maintain relatively high fault detection rates, from 50% to 97% as reported in [17]. Thus, we employ pairwise testing [18] in SBI. A set of pairwise tests PT_i is required to cover all interactions of each pair of parameters for each test API command ac_i, such that each test case in PT_i contains i, v values, one for each parameter in ac_i. In other words, for each pair of parameter values $apv_{j,a}$ and $apv_{k,b}$, where $apv_{j,a} \in ap_j$ and $apv_{k,b} \in ap_k$, there exists at least one test in PT_i that contains both $apv_{j,a}$ and $apv_{k,b}$ [14]. The set of pairwise tests required to cover all the pairwise interactions of each parameter pair for all the test API commands in the test suite is:

$$PT_T = \bigcup_{i=1}^{u} PT_i \qquad (2)$$

where, u is the number of test API commands covered by T, and PT_i is the set of pairwise tests required for test API command i.

Based on the above-mentioned notations, we define test case implantation as: automatically modifying and/or deleting existing statements from the original test suite and/or adding new statements, with the aim to construct a new test suite, which meets a set of predefined criteria, e.g., maximizing the pairwise coverage of parameter values of test API commands. Notably, the defined test case implantation does not increase the total number of test cases as compared with the original test suite. We use the following function to represent implanting t_a into $t_{a'}$:

$$Implant(t_a) \rightarrow t_{a'} \qquad (3)$$

4.2 Problem Representation

Let $S = \{s_1, s_2, \dots, s_{ns}\}$ represents a set of potential solutions, where $ns = 2^p - 1$ and $p = \sum_{t \in T} n(ST_t)$. Each solution $s = \{t_1, t_2, \dots, t_n\}$ has the same number of test cases as the original test suite T, and some of the test cases from T are chosen for implantation. $Cost = \{cost_1, \dots, cost_{ncost}\}$ refers to a set of cost measures (e.g., execution time of the test suite) and $Effect = \{effect_1, \dots, effect_{neffect}\}$ denotes a set of effectiveness measures (e.g., coverage of configuration variable values) for evaluating the quality of a solution.

Problem: Search a solution s_f from the total number of ns solutions in S that can achieve the maximum effectiveness with minimum cost.

$$\forall_{i=1\ to\ neffect} \forall_{j=1\ to\ ns}\ Effect\ (s_f, effect_i) \geq Effect\ (s_j, effect_i)$$
$$\cap\ \forall_{k=1\ to\ ncost} \forall_{j=1\ to\ ns}\ Cost\ (s_f, cost_k) \leq Cost\ (s_j, cost_k)$$

$Effect\ (s_j, effect_i)$ returns the i^{th} effectiveness measure of s_j, and $Cost\ (s_j, cost_k)$ returns the k^{th} cost measure of s_j.

4.3 Cost and Effectiveness Measures

This section formally defines three cost measures (Section 4.3.1) and two effectiveness measures (Section 4.3.2) based on the problem defined in Section 4.2.

4.3.1 Cost Measures

Number of implanted test cases (*NIT*): Since not all the test cases in the original test suite are selected for implantation, we define NIT to measure the total number of test cases chosen by the search for implantation, which can be calculated as the total number of test cases that exist in s but not in the original test suite T. The number of implanted test cases can be calculated as:

$$NIT = n(\bigcup_{t \in s} t : t \notin T) \qquad (4)$$

Our aim is to minimize NIT so that changes are introduced to a minimum number of the existing test cases for simplifying maintenance [19, 20].

Number of changed statements (*NCS*): A statement is called a changed statement if an existing statement is modified or removed or a new statement is added to the test case. The number of changed statements in a solution s is the sum of the numbers of modified statements (MST), deleted statements (DST), and added statements (AST) for each test case in s, such that:

$$NCS = \forall_{t \in s}(n(MST_t) + n(DST_t) + n(AST_t) : t \notin T) \qquad (5)$$

If $t_{a'}$ is the implanted test case for the original test case t_a:

$$MST_{ta'} = \forall_{tm \in ta'} \uplus_{st \in tm} st \notin t_a, DST_{ta'} = \forall_{tm \in ta} \uplus_{st \in tm} st \notin t_{a'} \text{ and}$$
$$AST_{ta'} = n(ST_{ta'}) - n(ST_{ta}) + DST_{ta'}.$$

We aim to minimize NCS to ensure that a statement is only changed when necessary (e.g., to cover more configuration variable values).

Estimated execution time (*EET*): The execution time of a solution (i.e., an implanted test suite) refers to the time required for executing all its test cases. The solutions update dynamically during search and many solutions are produced, which makes it difficult to execute the solutions for getting their execution time. For example, 25000 implanted test suites produced by search have to be executed 25000 times when the number of fitness evaluation is set as 25000, which is computationally expensive and infeasible. Thus, we propose to statically estimate the execution time of a test case in the solution based on the execution time of each statement of the test case. We measure the average execution time for each statement in a test case t_j ($ETES_j$) using the historical execution time of the test case: $ETES_j = \frac{et_j}{n(ST_j)}$, where et_j is the historical execution time of the statement and $n(ST_j)$ is the number of statements included in t_j. The estimated execution time of the test cases in the solution s is:

$$EET = \sum_{t \in s} n(ST_t) \times ETES_t \qquad (6)$$

where, $n(ST_t)$ is the total number of statements in a test case. Our aim is to minimize the estimated execution time of the test cases included in a solution.

To ensure that *EET* is accurate for estimation, we conducted a pilot experiment by producing 20 implanted test suites using our approach, executing them and comparing the real execution time with the estimated execution time (i.e., *EET*). We noticed that the difference between the real execution time and *EET* was on average 395 seconds, which has no practical differences. Therefore, we used *EET* to estimate the real execution time in our context.

4.3.2 Effectiveness Measures

Number of configuration variable values covered (*NCVV*): Based on equation 1, the set of configuration variable values covered by a solution s is: $CCVV = \bigcup_{cv \in CV} F(cv)$, where $F(cv)$ is a function that returns the set of configuration variable values for cv. The number of configuration variable values covered by s can be calculated as:

$$NCVV = n(CCVV) \quad (7)$$

For example, for the sanitized test case in Table D-1, $NCVV$ is one as it covers one configuration variable with one value (i.e., *packetlossresilence* with the value *off*). The goal is to achieve the maximum coverage of configuration variable values.

Pairwise coverage of parameter values of test API commands (*PCPV*): $PCPV$ is defined to measure how much pairwise coverage of parameter values of test API commands can be covered by a solution s, and it is calculated as below:

$$PCPV = \forall_{ac \in s} \forall_{i=1}^{n(F(ac))} \forall_{j=i+1}^{n(F(ac))} \left(\sum n(F(ap_i)) \times n\big(F(ap_j)\big) \right) \quad (8)$$

such that $n(F(ac)) > 1$, where $F(ac)$ is a function that returns the set of API parameters for the test API command ac, while $F(ap_i)$ and $F(ap_j)$ are functions that return the set of values in the test API parameters i and j, respectively. For example, for a solution with only one test case (i.e., sanitized test case in Table D-1), $PCPV$ is six since the test API command dial covers six pairs of parameter values. The goal is to maximize the pairwise coverage of parameter values of test API commands.

Each objective function measures the quality of a solution from a particular user perspective. For example, objective function $NCVV$ measures the quality of a solution in terms of the coverage of configuration variable values. These objectives are usually independent of one another. For instance, NIT measures the number of implanted test cases while NCS measures the number of changed statements. If all the test cases in a test suite are implanted, it has a maximum possible value for NIT but it could have a low value of NCS if the number of changed statements is few in each test case. Similarly, if a lot of changes are introduced in a few test cases, it could have a high value for NCS but not for NIT. Moreover, EET measures the execution time of a test case, which is independent of NCS and NIT. Additionally, $NCVV$ measures the number of configuration variable values tested by a test case while $PCPV$ measures the pairwise coverage of parameter values of

test API commands, which are different from the other objectives and also with each other (Sections 3 and 4).

5 SBI: Search-Based Test Case Implantation Approach

This section first provides an overview of SBI (Section 5.1) followed by the detailed presentation of *test case analyzer* (Section 5.2), and *test case implanter* (Section 5.3).

5.1 Overview of SBI

Fig. D-2 presents an overview of SBI consisting of two key components: *test case analyzer* and *test case implanter*. The *test case analyzer* component ensures that implanted test cases are semantically correct (e.g., two new statements should be added in a correct order). For this, the *test case analyzer* component statically analyzes each test case in the original test suite to obtain the program dependence graph [21, 22] for each *test method*, which is required to know dependences among statements. For example, on removing one statement another dependent statement(s) also need to be removed in the *test method* (see Section 5.2). Nodes in the program dependence graph represent the statements in the *test method* and edges represent the control and data dependence edges [23]. Moreover, the *test case analyzer* component automatically classifies all statements into the six categories defined in using the statement information document (created one time), which is described in detail in Section 5.2.

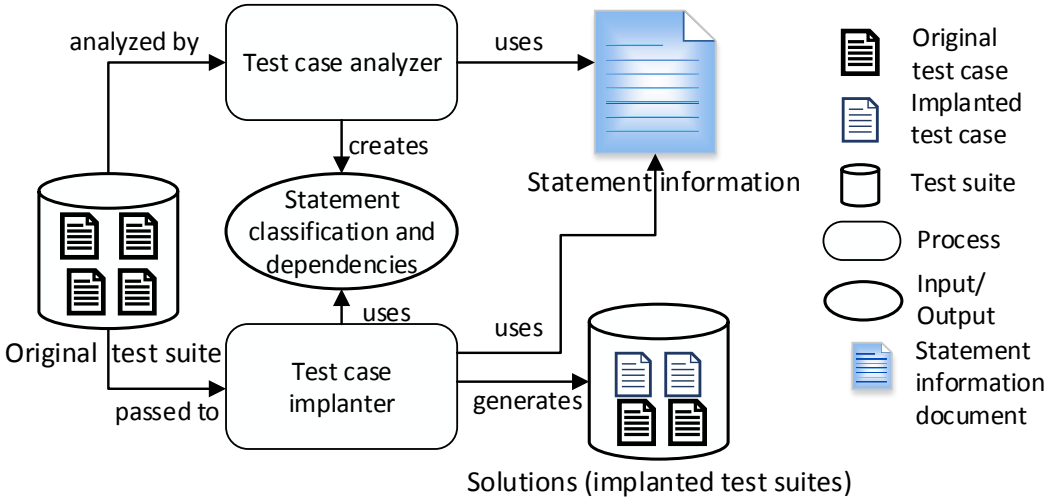

Fig. D-2: Overview of SBI

As depicted in Fig. D-2, the original test suite is passed to the *test case implanter* component to generate solutions by changing the values of the configuration variables and test API parameters from the list of available values provided in the statement information document (detailed in Section 5.3). Each generated solution includes implanted test cases and remaining (unchanged) test cases in the original test suite that are not chosen for implantation. For implanting a test case, changes are made to one or more of its classified statement (recall Table D-2) using the *test case implanter* component and program

dependence graph produced by *the test case analyzer* component is used to change the affected statements.

5.2 Test Case Analyzer

For each test case in the original test suite, the *test case analyzer* component automatically classifies all the statements of the test case into the six categories (Table D-2) and constructs a program dependence graph for each *test method*.

Statement classification. The *test case analyzer* component uses the statement information document (Fig. D-2) for classifying the statements in the test case. Generally, the statement information document includes: 1) *keywords* to distinguish between the different statements (e.g., for the sanitized test case in Section 3, *packetlossresilence, dial*, *wait,* and *assert* are defined as *keywords* to differentiate *configuration*, *execution*, *wait*, and *assertion statements*, respectively), 2) allowed values for the variables/parameters (that the test engineers need to test), and 3) domain specific rules for identifying the dependency between statements (e.g., later statement(s) in a test case may use same values for the same variable as the preceding statement). Notice that test engineers can build such statement information document based on their specific testing practice in any format (e.g., XML in our case).

In our context, the list of configuration variable names and test API commands are specified in the statement information document to differentiate between *configuration* and *execution statements*. Moreover, *assertion* and *wait statements* are classified based on whether they contain the keyword "assert" or "wait", respectively. The *assignment* statement is classified based on if they are used as a value at the 1) configuration variable or 2) test API parameter/s (e.g., *callrate_var* at line 3 in Table D-1 is used as a value in the parameter *callrate* in line 4). The program dependence graph is created using data and control dependences between statements, as explained below.

Data dependence. There exists data dependence between two statements if the second statement refers to the data of the first statement [22]. We define two sets of data dependences: 1) general and 2) domain specific. General dependences apply to all contexts, whereas domain specific data dependences are specific to a particular domain. There exists general data dependence between two statements in a *test method* if a variable in one statement has an incorrect value when the two statements are reversed. For example, as shown in Table D-1, the statement in line 4 is data dependent on the statement in line 3 as parameter *callrate* in *dial* is defined in line 4 (i.e., *callrate_var*). We use domain specific rules defined explicitly in the statement information document (as illustrated in Fig. D-2) to create domain specific data dependence. For example, in the context of the video conferencing company if there exist two or more *execution statements* such that the test API command in the *execution statements* use one or more same parameters (e.g., the test API commands *dial* and *call_transfer* have the same parameter *protocol*) the value of the parameter in the test API command in the second *execution statement* must take the

same value as the same parameter defined in the test API command at the first *execution statement*.

Control dependence. Similar to data dependences, there exist general and domain specific control dependences. There exists a general control dependence between two statements if the value of the first statement controls the execution of the second statement [22]. For example, in the sequence, **if** (*protocol = SIP*) **then** *accept,* the statement *accept* depends on the predicate statement **if** (*protocol = SIP*) since the value of *protocol* determines if *accept* is executed. As in data dependence, domain specific control dependence is based on domain specific rules based on the statement information document (as illustrated in Fig. D-2). For example, in the context of the video conferencing company, if there are two *execution statements* (e.g., one with test API command *dial* and the other with *call_transfer*), the execution of the second *execution statement* (i.e., *call_transfer*) depends on the execution of the first *execution statement*. To capture this dependence, we keep track of the statement execution order at the *test method* in the test case.

5.3 Test Case Implanter

The *test case implanter* component includes two steps: *test case selection* and *test case implantation*. The first step is to select test cases from the original test suite for implantation. To this end, we use decision variables (Section 5.3.1) to guide the search for selecting test cases based on the defined fitness function. The second step is to implant the selected test cases by changing statement(s) (e.g., adding new statement) in *test methods* using a *mutation operator* with three *operations* (i.e., *addition, modification,* and *deletion*).

5.3.1 Solution Representation

We represent each solution at two different levels: test suite level and test case level, as shown in Fig. D-3. At the test suite level, test cases in solution s are represented with an array of real variables, $V = \{v_1, \dots, v_n\}$, where each variable v_i is associated with test case t_i (Fig. D-3). The value of v_i ranges from 0 to 1, and this value indicates whether t_i is selected for implantation in s. A value greater than 0.5 indicates that the test case is selected for implantation, while a value less than or equal to 0.5 indicates otherwise. Initially, each variable in V (i.e., v_i) is assigned a random value from 0 to 1, and during the search, the *test case implanter* component of SBI returns the solutions guided by the fitness functions defined in the next section.

Test Case	t_1	t_2	t_3	…	t_{n-2}	t_{n-1}	t_n
Variable	v_1	v_2	v_3	…	v_{n-2}	v_{n-1}	v_n

Test Suite Level

Test Method	$tm_{i,1}$	…	$tm_{i,o}$
Statements	$st_{i1,1}, \dots, st_{i1,q}$	…	$st_{io,1}, \dots, st_{io,q}$

Test Case Level for Test Case t_i

Fig. D-3. Two different levels in a solution

At the test case level, a particular test case is composed of a number of *test methods* and each *test method* includes a set of statements. Fig. D-3 depicts a set of *test methods* ($tm_{i,j}$) for test case t_i with the total number of *test methods* being i, o.

5.3.2 Fitness Function

Recall that the goal of SBI is to cost-effectively implant existing test cases, while 1) maximizing the effectiveness (i.e., $NCVV$ and $PCPV$) and 2) minimizing the cost (i.e., NIT, NCS, and EET). With this goal in mind, we have defined the five cost-effectiveness measures in Section 4.3. Since values of different cost and effectiveness measures are not comparable, we use the normalization function suggested in [24] to normalize the values in the same magnitude of 0 and 1 for all the five cost and effectiveness measures: $Nor\big(F(x)\big) = \frac{F(x)}{F(x)+1}$, where $F(x)$ is a function for NIT, NCS, EET, $NCVV$, and $PCPV$ (equations 4 - 8). Thus, for the five cost and effectiveness measures (equations 4 – 8), we define the following five objective functions:

$$F(O_1) = Nor(NIT), F(O_2) = Nor(NCS), F(O_3) = Nor(EET)$$
$$F(O_4) = 1 - Nor(NCVV), F(O_5) = 1 - Nor(PCPV)$$

Note that we define our multi-objective search problem as a minimization problem, i.e., a solution with a lower value for an objective implies a better performance of a solution. Therefore, we subtracted 1 for the effectiveness measures: $F(O_4)$ and $F(O_5)$.

5.3.3 Test Case Implantation

Test case implantation occurs at the test case level (Fig. D-3). For this, values of configuration variables in *configuration statements* or values of parameters in *execution statements* are based on their allowed values provided in the statement information document (Fig. D-2). When changing values of parameters, the pairwise testing strategy is applied as explained in Section 4.1. Recall that the statement classification is provided by the *test case analyzer* component (Section 5.2). When a particular statement in a test method is changed, forward and backward slicing [25] is applied to obtain affected statements of the test method (using the program dependence graph from the *test case analyzer* component) that should be changed as well. A slice refers to a set of statements that influence the value of a variable at a particular program location (i.e., the location of the changed statement in this context) [26]. The process is described in detail in the next section.

Search Operators at the Test Suite Level: The *test case analyzer* component integrates the defined fitness function into a multi-objective search algorithm (e.g., NSGA-II [3]). We chose the widely used tournament selector [3] as the selection operator to select individual solutions with the best fitness for inclusion into the next generation. The crossover operator is applied at the test suite level, which randomly swaps parts of two parent solutions (i.e., test suites) to produce two offspring solutions. To this end, we chose a single point

crossover operator that randomly selects the same point in both the parent solutions for generating the offspring solutions as shown in Fig. D-4.

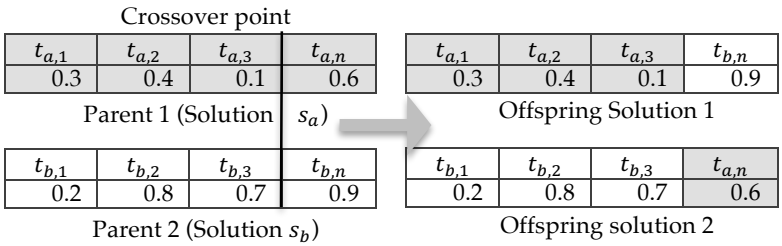

Fig. D-4. Single Point Crossover applied between Two Solutions

The generated offspring solutions contain the test cases and the variable values associated with the test cases from the parent solutions as shown in Fig. D-4. Note that we do not apply the crossover operator at the test case level since the *setup class*, *teardown*, and *teardown class* required for running the test methods may vary across test cases, which might lead to semantically incorrect test cases.

We apply the *mutation operator* at both the test suite and test case levels (Fig. D-3). In terms of the test suite level, the *mutation operator* is defined to randomly swap the values of two variables (Section 5.3.1) based on the pre-defined mutation probability (e.g., $1/(size\ of\ the\ test\ suite)$), and recall that each variable represents a test case in our context. After the values of the variables have been swapped, if the value of the variable is greater than 0.5, the test case represented by the variable is selected for implantation. For example, in Fig. D-4, if the *mutation operator* is applied to swap the values of the variables representing $t_{a,1}$ and $t_{b,n}$ in the offspring solution 1, the variable representing $t_{a,1}$ will have a new value 0.9 while the variable representing $t_{b,n}$ will have the value 0.3. This causes $t_{a,1}$ to be selected for implantation instead of $t_{b,n}$.

Mutation Operator at the Test Case Level: With respect to the test case level (Fig. D-3), we defined three operations for the mutation operator: modification, addition, and deletion inspired from the work in [27, 28]. Each operation is randomly chosen with a probability of $1/3$ for each test case selected for the implantation. Therefore, on average, at least one operation is applied to the selected test case for implantation. Moreover, for a test case t_i with the number of test methods as o, i.e., $t_i = \{tm_{i,1}, \dots, tm_{i,o}\}$ in , each test method is mutated with a probability $1/o$ using the chosen operation (e.g., addition) for the mutation operator. Note that the operation is applied to the *configuration* or *execution statement* in a test case (Table D-2) because they determine the functionality of the SUT that is being tested. If tm_c is the test method to be changed in the test case t_i with the number of configuration and execution statements e, each configuration or execution statement is mutated with a probability of $1/e$. Suppose st_c is the statement to be changed, we explain the three operations for the mutation operator below.

Modification operation. The value of the configuration variable or parameter for the test API command in st_c is randomly changed to cover an uncovered 1) value of the configuration variable for *configuration statement* or 2) pairwise coverage of parameter

values of the test API commands for *execution statement*. After the statement st_c is modified, if there exists statement(s) dependent on st_c (Section 5.2), they are also modified using the statement dependencies obtained from the *test case analyzer* component (Section 5.2). For example, in Table D-1 if the *modification operation* is applied to the *execution statement* at line 4, i.e., the test API command *dial*, the values of the parameters are randomly changed to increase the pairwise coverage. If the parameter *callrate* in *dial* (pointing to *callrate_var*) is required to be modified from *6000* to *64*, the variable *callrate_var* is modified to *64*.

Addition operation. A copy of statement st_c (i.e., $st_{c'}$) is created with the random uncovered 1) value of the configuration variable for *configuration statement* or 2) pairwise coverage of parameter values of the test API commands for *execution statement*. The new statement $st_{c'}$ is then added to a new *test method* $tm_{c'}$, and $tm_{c'}$ is filled with all the statements dependent on st_c in tm_c (Section 5.2). If the values of the statement(s) depend upon $st_c{}'$, the dependent statement (s) are also modified after adding to the new *test method*. In the running example in Table D-1, if the configuration variable *packetlossresilence* at line 2 is selected for applying the *addition operation*, *test case analyzer* will add a new statement *packetlossresilence* with the uncovered value (i.e., *on*) in a new *test method*. Since the statements in lines 4, 5, and 6 are control dependent on line 2 (Table D-1), they are also added in the new *test method*. Moreover, the statement in line 4 is also added in the new *test method* since it is data dependent on line 3 (Table D-1).

Deletion operation. A statement st_c is deleted from *test method* tm_c if the values of the configuration variables or the parameters (for the pairwise coverage of parameter values of test API commands) tested by tm_c have been already covered by other test cases in solution s. For example, if the *deletion operation* is applied in line 4 at Table D-1 (i.e., statement with *dial*) and the parameter values for *dial* (line 4 in Table D-1) are covered by another test case in solution s, then line 4 is removed from the *test method* in Table D-1. Moreover, lines 2, 3, 5, and 6 are dependent on line 4 in Table D-1, and therefore removed.

6 Experiment Design

In this section, we describe the experiment design (as shown in Table D-3), which includes the case studies (Section 6.1), research questions (Section 6.2), evaluation metrics (Section 6.3), and statistical tests along with the experiment settings (Section 6.4).

6.1 Case Studies

To evaluate SBI, we chose one industrial case study from the video conferencing company referred as CS_1, and the open source case study of SafeHome [4] referred as CS_2. The industrial case study focuses on automated testing of large-scale VCSs developed by the video conferencing company. Each VCS has an average of three million lines of embedded

C code and requires a thorough testing before releasing them to the market. We chose a test suite containing 118 test cases for evaluation, where on average, each test case consists of 4 *test methods* and 30.8 statements (as defined in Section 3).

Moreover, the SafeHome case study was constructed based on the open source implementation of a home security and surveillance system [29], which consists of in total 13 Java classes. Each class has on average 263.4 lines of Java code and the detailed description of these classes can be consulted in [4]. Notice that the original implementation reported in [29] includes only 9 configuration variables (i.e., 6 Boolean variables and 3 Integer variables) and lacks sufficient parameters to evaluate SBI (i.e., most of the test API commands have 0 or 1 parameter). Therefore, we extended the SafeHome case study by adding: 1) additional 19 configuration variables that include 8 String variables with on average 5 values to configure for each, 2 Integer variables, and 9 Boolean variables and 2) in total 37 methods in the source code (e.g., *createUser*). 3,424 lines of non-comment Java source code (calculated using sloccount [30]) were added in total for the case study.

To obtain the original test suite for implantation for the SafeHome case study, we applied EvoSuite [31] to automatically generate in total 94 test cases (as the original test suite for implantation) including an average of 2.4 *test methods* and 19 statements for each test case. Note that our aim is to implant the original test suite for increasing the configuration coverage rather than comparing the performance between SBI and EvoSuite. To make the experiment reproducible, we have made the extended SafeHome case study publically available at [32].

Table D-3. An Overview of the Experiment Design*

RQ	Task	Comparison	Case Study	Evaluation Metric	Statistical Test
1	$J_{1.1}$	$SBI_{NSGA\text{-}II}$, SBI_{RS}, and SBI_{WGA} with the original test suite	CS_1, CS_2	$NCVV$, $PCPV$	One-sample Wilcoxon Test
	$J_{1.2}$			EET, NIT, NCS	
	$J_{1.3}$		CS_2	SC, BC	
	$J_{1.4}$			MS	
2	$J_{2.1}$	$SBI_{NSGA\text{-}II}$, SBI_{RS}, and SBI_{WGA}	CS_1, CS_2	$NCVV$, $PCPV$, EET, NIT, NCS	Vargha and Delaney, Mann-Whitney U Test
	$J_{2.2}$	$SBI_{NSGA\text{-}II}$ and SBI_{RS}		HV	
	$J_{2.3}$	$SBI_{NSGA\text{-}II}$, SBI_{RS}, and SBI_{WGA}	CS_2	SC, BC	
	$J_{2.4}$			MS	

*$NCVV$: number of configuration variable values covered, $PCPV$: pairwise coverage of parameter values of test API commands, EET: estimated execution time, NIT: number of implanted test cases, NCS: number of changed statements, SC: statement coverage, BC: branch coverage, MS: mutation score, HV: hypervolume.

6.2 Research Questions

To evaluate SBI, we aim at addressing the following two research questions (RQs).

RQ1. Sanity Check: This research question aims to compare the three SBI variants using NSGA-II (i.e., $SBI_{NSGA\text{-}II}$), RS (SBI_{RS}), and weight-based GA (i.e., SBI_{WGA}) against the original test suite. This RQ is divided into four sub RQs:

RQ1.1 Effectiveness. Can SBI significantly increase 1) the coverage of configuration variable values and 2) pairwise coverage of parameter values of test API commands (Section 4.3.2)?

RQ1.2 (Acceptability). Can test suites implanted by the three variants of SBI maintain an acceptable cost in terms of estimated execution time, number of changed statements, and number of implanted test cases (defined in Section 4.3.1)?

RQ1.3 (Coverage). Can SBI significantly increase the code coverage in terms of statement coverage (SC) and branch coverage (BC)?

RQ1.4 (Mutation Analysis). Can the implanted test suites produced by SBI significantly improve the mutation score as compared with the original test suite? We performed mutation analysis to further assess the fault detection capability achieved by SBI. Since we do not have access to the source code of the industrial case study (i.e., CS_1) due to confidential concerns, RQ1.3 and RQ1.4 are only addressed using CS_2.
If SBI passes the sanity check, the next step is to check which variant of SBI performs the best, which forms RQ2.

RQ2. Comparison across different variants of SBI: This RQ aims to find the best SBI variant. It is further divided into four sub RQs.

RQ2.1 (Cost-Effectiveness). Which SBI variant has the best performance in terms of cost-effectiveness (Section 4.3)?

RQ2.2 (Overall Quality of Solutions). Between $SBI_{NSGA\text{-}II}$ and SBI_{RS}, which SBI variant produces the best overall quality solution? Note that SBI_{WGA} is not considered in this RQ since SBI_{WGA} combines all the objectives into a single objective and only produces one solution at one run unlike $SBI_{NSGA\text{-}II}$ and SBI_{RS}, each of which produce 100 solutions (i.e., defined as the *population size*) at each run.

RQ2.3 (Coverage). Which SBI variant has the highest statement coverage (SC) and branch coverage (BC)?

RQ2.4 (Mutation Analysis). Which SBI variant has the highest mutation score (MS)?

6.3 Evaluation Metrics

We used the evaluation metrics $NCVV$ (equation 7) and $PCPV$ (equation 8) to measure the effectiveness, and EET, NIT and NCS (equations 4 – 6) to measure the cost of the test suites using tasks $J_{1.1}, J_{1.2}$, and $J_{2.1}$ (Table D-3). Tasks $J_{1.3}$ and $J_{2.3}$ were conducted to measure the code coverage with evaluation metrics SC and BC. Specifically, SC measures the number of statements in the source code that are executed when executing a given test suite, while BC measures the number of possible branch(es) from each decision point that is executed [33]. RQ1.4 and RQ2.4 were addressed by tasks $J_{1.4}$ and $J_{2.4}$ using the mutation score (MS) [34] as the evaluation metric, which is the ratio of killed mutants out of the

total number of non-equivalent mutants [34], and it has been widely used to measure the fault detection capability of test suites [35-38].

It is common to apply quality indicators such as *hypervolume* (*HV*) to compare the overall performance of multi-objective search algorithms [39, 40]. Therefore, we used *HV* to compare the overall performance of $SBI_{NSGA\text{-}II}$ and SBI_{RS} based on the guidelines provided in [41] (task $J_{2.2}$). Specifically, *HV* calculates the volume in the objective space covered by a non-dominated set of solutions (e.g., Pareto front), which is considered as a combined measurement of both convergence and diversity [42]. We however did not compare *HV* for the weight-based GA since it converts the multi-objective problem into a single objective and produces only one solution for each run.

6.4 Statistical Tests and Experiment Settings

6.4.1 Statistical Tests

To choose an appropriate statistical test, we first performed the Shapiro-Wilk test [43, 44] to assess whether the data samples produced are normally distributed. The results of the Shapiro-Wilk test showed the obtained data samples were not normally distributed, and thus we chose the one-sample Wilcoxon test as recommended in [45] to statistically evaluate results of RQ1 (i.e., RQ1.1 - RQ1.4), i.e., tasks $J_{1.1}$ - $J_{1.4}$ (Table D-3). We used the one-sample Wilcoxon test (p-value) since the coverage of the original test suite (e.g., $NCVV, SC$) is fixed. Moreover, we compare mean values for the coverage of the original test suite and the test suites implanted by SBI to see in which direction the results are significant, i.e., which approach is better when the p-value is less than 0.05.

Moreover, we used the Vargha and Delaney $\hat{A}_{12}$ statistics [46] and Mann-Whitney U test [47] to statistically evaluate the results of RQ2 (i.e., RQ2.1 – RQ2.4), i.e., tasks $J_{2.1}$ - $J_{2.4}$ (Table D-3), based on the guidelines in [45]. The Vargha and Delaney statistics is defined as a non-parametric effect size measure and evaluates the probability of yielding higher values for each evaluation metric for two algorithms A and B. Additionally, the Mann-Whitney U test is used to indicate if observations (e.g., objective values) in one data sample are likely to be larger than observations in another sample, and p-value was used to check if the result is significant. For all the statistical tests, we considered a p-value below 0.05 as statistically significant, a commonly used threshold in SBSE studies [45]. For two algorithms A and B, A has significantly better performance than B if $\hat{A}_{12}$ is higher than 0.5 and the p-value is less than 0.05.

6.4.2 Experiment Settings

SBI is implemented using a Java framework jMetal [48], which has been widely used for various multi-objective optimization problems [49-51]. We use the same *population size* (i.e., 100) in all the algorithms (i.e., NSGA-II, RS, and WGA), which is the standard setting in jMetal. Moreover, we tuned *crossover rate* and *mutation rate* at the test suite

level using the iRace optimization package [52], which has been widely used in literature for automatic algorithm configuration [53-56]. Table D-4 presents the parameter settings (*crossover rate* and *mutation rate*) for NSGA-II and WGA for CS_1 and CS_2. Note that RS does not involve crossover and mutation. Additionally, we set the maximum number of fitness evaluations (i.e., termination criterion) as 25,000 for all the algorithms across the two case studies.

Table D-4. Parameter Settings for NSGA-II and Weight-based GA

Algorithm	Case Study	Crossover Rate	Mutation Rate
NSGA-II	CS_1	0.72	0.31
	CS_2	0.78	0.24
WGA	CS_1	0.74	0.25
	CS_2	0.80	0.17

SBI with NSGA-II (i.e., $SBI_{NSGA\text{-}II}$) and RS (i.e., SBI_{RS}) produce 100 solutions (equal to *population size*) for each run in each case study, while SBI with weight-based GA (i.e., SBI_{WGA}) produce only one solution for each run as discussed in Section 2.2. In practice, the decision maker(s) selects the solution from the final Pareto front based on their preference of the objective. For instance, if the decision maker(s) values pairwise coverage of parameter values of test API commands ($PCPV$) higher than the other objectives, he/she selects solutions with a higher value from the final Pareto front. Since it is not fixed which solution is actually picked by the decision maker(s), we have compared the quality of all the generated solutions for all the algorithms with SBI.

Regarding *SC* and *BC* (RQ1.3 and RQ2.3), we used the open source tool Eclemma [57] to measure the *SC* and *BC* achieved by the implanted test suites and the original one. For mutation analysis (RQ1.4 and RQ2.4), we used the Java-based mutation tool PIT [58], which has been extensively used in mutation testing [59, 60]. All the seven basic *mutation operators* in PIT (i.e., conditionals boundary, increments, invert negatives, math, negate conditionals, return values, and void method calls) were applied and 1594 non-equivalent mutants were generated for CS_2. In addition, we ran SBI 10 times to account for the random variation for each case study.

7 Results, Analysis, and Overall Discussion

Results and analysis are presented in Sections 7.1 and 7.2, followed by the overall discussion (Section 7.3).

7.1 RQ1. Sanity Check

7.1.1 RQ1. Effectiveness

Recall that RQ1.1 aims to assess the effectiveness of the implanted test suites produced by the three SBI variants ($SBI_{NSGA\text{-}II}$, SBI_{RS}, and SBI_{WGA}) in terms of the two effectiveness measures: the number of configuration variable values covered ($NCVV$) and the pairwise

coverage of parameter values of test API commands ($PCPV$), as described in Section 4.3.2. Table D-5 summarizes the values for the different evaluation metrics achieved by the SBI variants for the implanted test suites (i.e., solutions) and the original test suites of the two case studies (i.e., CS_1 and CS_2). Recall from Section 6.4.2 that SBI is executed 10 times, and each run produces 100 optimal solutions for $SBI_{NSGA\text{-}II}$ and SBI_{RS}, and one optimal solution for SBI_{WGA}.

As shown in Table D-5, test suites generated by all the SBI variants have better values for $NCVV$ and $PCPV$. Specifically, the mean differences between the values for $NCVV$ and $PCPV$ from the original test suites and the implanted test suite produced by 1) $SBI_{NSGA\text{-}II}$ are 15.3 and 100.8 for CS_1, and 11.4 and 158.3 for CS_2, 2) SBI_{RS} are 6.8 and 74.3 for CS_1, and 5.0 and 29.6 for CS_2, and 3) SBI_{WGA} are 6.4 and 65.5 for CS_1, and 5.3 and 13.8 for CS_2. Moreover, all the mean differences are statistically significant since all the p-values are less than 0.05 (from the one-sample Wilcoxon test), which shows that all the SBI variants managed to perform significantly better than the original test suite regarding $NCVV$ and $PCPV$.

Table D-5. Results of Evaluation Metrics for the Original Test Suites and Implanted Test Suites*

CS	Test Suite	*NCVV*		*PCPV*		*EET*		*NIT*		*NCS*		*SC*		*BC*	
		Mean	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**
CS_1	Original	86.0	N/A	212.0	N/A	8222.0 m	N/A								
	$SBI_{NSGA\text{-}II}$	101.3	10.3	312.8	72.9	8368.0m	657.2	39.9	16.1	733.7	942.9	N/A			
	SBI_{RS}	92.8	2.6	286.3	18.8	8305.0m	77.5	19.7	4.5	88.5	51.6				
	SBI_{WGA}	92.4	1.6	277.5	15.1	8340.0m	53.5	13.8	1.3	54.8	37.9				
CS_2	Original	55.0	N/A	238.0	N/A	3.5s	N/A					66.7	N/A	49.9	N/A
	$SBI_{NSGA\text{-}II}$	66.4	5.9	396.3	123.2	3.8s	0.3	33.3	15.7	263.8	295.9	71.7	3.6	57.8	5.7
	SBI_{RS}	60.0	1.4	267.6	14.6	3.5s	0.0	15.3	4.2	43.8	13.5	67.5	0.6	51.3	1.0
	SBI_{WGA}	60.3	0.7	251.8	6.0	3.6s	0.0	11.3	2.7	22.7	4.3	67.2	0.4	50.9	0.8

**CS*: case study, *NCVV*: number of configuration variable values covered, *PCPV*: pairwise coverage of parameter values of test API commands, *EET*: estimated execution time, *NIT*: number of implanted test cases, *NCS*: number of changed statements, *SC*: statement coverage, *BC*: branch coverage, SD: standard deviation, m: minutes, s: seconds.

Since the results for CS_1 and CS_2 are consistent and all the implanted test suites have significantly higher $NCVV$ and $PCPV$ as compared to the original test suites, we can answer RQ1.1 as: the implanted test suites achieved significantly higher effectiveness than the original one, which demonstrates that SBI is effective. Moreover, the test suites implanted by the three SBI variants managed to achieve on average 11.0% higher $NCVV$ and 37.8% higher $PCPV$ for CS_1, and 13.1% and 28.2% higher $PCPV$ for CS_2.

7.1.2 RQ1.2. Acceptability

In terms of EET, we can observe from Table D-5, that on average the implanted test suites produced by $SBI_{NSGA\text{-}II}$, SBI_{RS}, and SBI_{WGA} require 1.4% more EET for CS_1 and 3.5% more EET for CS_2. Moreover, in terms of NIT, the implanted test suites produced by the SBI variants modified 20.7% and 21.3% test cases for CS_1 and CS_2. Finally, in terms of NCS, the implanted test suites produced by the SBI variants modified on average 8.0%and 6.2% statements for CS_1 and CS_2. Additionally, all the mean differences for EET, NIT, andNCS

are statistically significant since the p-values are less than 0.05 (obtained from the one-sample Wilcoxon test). Therefore, we conclude that SBI can maintain acceptable cost without largely increasing the test case execution time indicating that SBI is cost-effective.

7.1.3 RQ1.3. Code Coverage

This RQ aims to evaluate whether SBI can increase the overall code coverage in terms of SC and BC using the SafeHome case study (i.e., CS_2). From Table D-5, we can observe that the mean differences between the values produced by the original test suite and the implanted one by 1) $SBI_{NSGA\text{-}II}$ are 5.0% and 7.9% for SC and BC, 2) SBI_{RS} are 0.8% and 1.4% for SC and BC, and 3) SBI_{WGA} are 0.5% and 1.0% for SC and BC. Additionally, all the mean differences are statistically significant since the p-values are less than 0.001 (obtained from the one-sample Wilcoxon test). Thus, we summarize that SBI can significantly increase the code coverage of the original test suite.

7.1.4 RQ1.4. Mutation Score

This RQ aims to check if the test suites implanted by SBI have higher mutation scores (MS) than the original test suite. Each execution of $SBI_{NSGA\text{-}II}$ and SBI_{RS} produces 100 optimal solutions (Section 6.4.2), and it is quite expensive to perform mutation analysis for all the 1000 solutions (produced by executing SBI 10 times) since it takes more than four minutes to perform mutation analysis for one solution. Thus, we chose only two solutions produced in each run of $SBI_{NSGA\text{-}II}$ and SBI_{RS} to perform mutation analysis. Based on the existing work [61, 62], we chose the solutions by following these two ways: 1) random, referred as *random solution* and 2) highest average value of all the defined cost-effectiveness measures (Section 4.3) referred as *selected solution*. Note that for SBI_{WGA} we perform mutation analysis for all the generated solutions since each run of SBI_{WGA} produces only one solution.

Table D-6. Mutation Scores for the Different Approaches*

Solution	*MS*	Mean difference compared to original	*p*-value
Original	33.90 %	N/A	
$SBI_{NSGA\text{-}II}$-RA	36.51 %	2.60 %	0.002
$SBI_{NSGA\text{-}II}$-SA	37.52 %	3.70 %	0.002
SBI_{RS}-RA	35.14 %	0.93 %	0.014
SBI_{RS}-SA	35.35 %	1.60 %	0.002
SBI_{WGA}	34.67 %	0.77 %	0.006

Table D-6 summarizes the results of *MS* for the original test suite, the average of 10 *random solutions* and 10 *selected solutions* produced by $SBI_{NSGA\text{-}II}$ and SBI_{RS}, and 10 solutions produced by SBI_{WGA}. Moreover, Table D-6 shows the results of the mean differences and the one-sample Wilcoxon test between the MS produced by the original test suite and 10 *random solutions* and 10 *selected solutions*. From Table D-6, we can conclude for RQ1.4 that the solutions implanted by SBI have a significantly higher MS since the p-values for the random solutions and selected solutions are less than 0.05 and

the mean difference is positive (e.g., 3.7% indicating that the *selected solutions* from $SBI_{NSGA\text{-}II}$ improved on average 3.7% MS as compared to the original one). Thus, we can answer RQ1.4 as SBI can detect more faults (as indicated by a higher MS)

7.2 RQ2. Comparison of $SBI_{NSGA\text{-}II}$ with the other SBI variants

7.2.1 RQ2.1. Cost-Effectiveness

In terms of effectiveness, it can be observed from Table D-7 that $SBI_{NSGA\text{-}II}$ produced the highest mean values for $NCVV$ and $PCPV$. Specifically, test suites implanted by $SBI_{NSGA\text{-}II}$ achieved a higher mean value for $NCVV$ than the test suites implanted by 1) SBI_{RS} for 9.9% and 11.8% for CS_1 and CS_2, and 2) SBI_{WGA} for 10.3% and 11.1% for CS_1 and CS_2. Moreover, test suites implanted by $SBI_{NSGA\text{-}II}$ have a higher mean value for $PCPV$ than the test suites implanted by 1) SBI_{RS} for 12.5% and 54.1% for CS_1 and CS_2, and 2) SBI_{WGA} for 16.7% and 60.7% for CS_1 and CS_2. From Table D-7, one can observe that $SBI_{NSGA\text{-}II}$ performed significantly better than 1) SBI_{RS} for $NCVV$ and $PCPV$ for both CS_1 and CS_2, and 2) SBI_{WGA} for $PCPV$ for CS_1 and both $NCVV$ and $PCPV$ for CS_2. Note that there is no significant difference in the performance for $NCVV$ for CS_1 between $SBI_{NSGA\text{-}II}$ and SBI_{WGA} (since p-value >0.05).

In terms of cost, test suites implanted by $SBI_{NSGA\text{-}II}$ had higher mean values for EET, NCS, and NIT. More specifically, (i) in terms of EET, test suites implanted by $SBI_{NSGA\text{-}II}$ required 1) 0.8% and 6.7% more EET than SBI_{RS} for CS_1 and CS_2, and 2) 0.4% and 6.4% more EET than SBI_{WGA} for CS_1 and CS_2; ii) in terms of NIT, test suites implanted by $SBI_{NSGA\text{-}II}$ had 1) 17.1% and 19.2% more NIT than SBI_{RS} for CS_1 and CS_2, and 2) 22.1% and 23.4% more NIT than SBI_{WGA} for CS_1 and CS_2; and iii) in terms of NCS, test suites implanted by $SBI_{NSGA\text{-}II}$ had 1) 17.8% and 12.3% more NCS than SBI_{RS} for CS_1 and CS_2, and 2) 18.7% and 13.5% more NCS than SBI_{WGA} for CS_1 and CS_2. Additionally, from Table D-7, one can observe that $SBI_{NSGA\text{-}II}$ performed significantly worse than SBI_{RS} and SBI_{WGA} for 1) NCS and NIT for CS_1, and 2) NCS, NIT, and EET for CS_2. There was no significant difference in the performance of $SBI_{NSGA\text{-}II}$ and SBI_{RS}, and $SBI_{NSGA\text{-}II}$ and SBI_{WGA} for EET for CS_1 (since p-value >0.05) as shown in Table D-7.

Table D-7. Comparison of the Cost-Effectiveness Measures using the Vargha and Delaney Statistics and U Test*

Evaluation Metric	$SBI_{NSGA\text{-}II}$ vs SBI_{RS}				$SBI_{NSGA\text{-}II}$ vs SBI_{WGA}			
	CS_1		CS_2		CS_1		CS_2	
	$\widehat{A}_{12}$	p-value	$\widehat{A}_{12}$	p-value	$\widehat{A}_{12}$	p-value	$\widehat{A}_{12}$	p-value
NCVV	0.56	**<0.05**	0.87	**<0.05**	0.62	0.21	0.97	**<0.05**
PCPV	0.77	**<0.05**	0.83	**<0.05**	0.79	**<0.05**	0.80	**<0.05**
NCS	0.24	**<0.05**	0.24	**<0.05**	0.15	**<0.05**	0.10	**<0.05**
NIT	0.13	**<0.05**	0.19	**<0.05**	0.02	**<0.05**	0.10	**<0.05**
EET	0.50	0.97	0.15	**<0.05**	0.54	0.69	0.24	**<0.05**

**NCVV*: number of configuration variable values covered, *PCPV*: pairwise coverage of parameter values of test API commands, *EET*: estimated execution time, *NIT*: number of implanted test cases, *NCS*: number of changed statements; the bold numbers in the table imply that the results are statistically significant.

Multi-objective search algorithm (e.g., NSGA-II) produce diverse solutions in the search space. Therefore, the implanted test suites have diverse values for the different objectives (as indicated by the high standard deviation in Table D-5). In order to check if $SBI_{NSGA\text{-}II}$ still manages to obtain better solutions (i.e., implanted test suites) than SBI_{RS} and SBI_{WGA} for solutions within the same space, we compare the implanted test suites with similar *EET* (Table D-8). From Table D-8, one can observe that for similar *EET*, test suites implanted by $SBI_{NSGA\text{-}II}$ still managed to achieve 1) 2.5% and 1.4% higher *NCVV* than SBI_{RS} for CS_1 and CS_2, and 2.9% and 0.4% higher *NCVV* as compared to SBI_{WGA}, and 2) 4.3% and 8.0% higher *PCPV* for CS_1 and CS_2 as compared to SBI_{RS}, and 11.6% and 11.8% higher *PCPV* as compared to SBI_{WGA}. Additionally, (i) in terms of *NIT*, the test suites implanted by $SBI_{NSGA\text{-}II}$ had 1) 6.6% and 8.0% more *NIT* than SBI_{RS} for CS_1 and CS_2, and 2) 11.6% and 11.8% more *NIT* than SBI_{WGA} for CS_1 and CS_2; and (ii) in terms of *NCS*, the test suites implanted by $SBI_{NSGA\text{-}II}$ had 1) 3.9% and 2.4% more *NCS* than SBI_{RS} for CS_1 and CS_2, and 2) 4.6% and 2.9% more *NCS* than SBI_{WGA} for CS_1 and CS_2.

To summarize, $SBI_{NSGA\text{-}II}$ manages to produce many diverse solutions with much higher effectiveness than SBI_{RS} and SBI_{WGA}. Moreover, for similar estimated execution time, the test suites implanted by $SBI_{NSGA\text{-}II}$ have higher effectiveness. Therefore, we conclude that SBI is more cost-effective than SBI_{RS} and SBI_{WGA}.

Table D-8. Results of Evaluation Metrics for Test Suites with similar Estimated Execution Time*

Case Study	Test Suite	*NCVV*	*PCPV*	*EET*	*NIT*	*NCS*	*SC*	*BC*
CS_1	$SBI_{NSGA\text{-}II}$	94.9	295.5	8305.0m	27.5	228.8	N/A	
	SBI_{RS}	92.8	286.3	8305.0m	19.7	88.5		
	SBI_{WGA}	92.4	277.0	8313.1m	13.9	62.3		
CS_2	$SBI_{NSGA\text{-}II}$	60.7	282.6	3.5s	22.8	75.2	69.2	54.7
	SBI_{RS}	60.0	267.6	3.5s	15.3	33.0	67.5	51.3
	SBI_{WGA}	60.5	252.5	3.5s	11.8	22.7	67.3	50.8

**CS*: case study, *NCVV*: number of configuration variable values covered, *PCPV*: pairwise coverage of parameter values of test API commands, *EET*: estimated execution time, *NIT*: number of implanted test cases, *NCS*: number of changed statements, *SC*: statement coverage, *BC*: branch coverage, m: minutes, s: seconds.

7.2.2 RQ2.2. Overall Quality of the Solutions

Recall that RQ2.2 is for check if $SBI_{NSGA\text{-}II}$ can manage to produce better overall quality solutions than SBI_{RS}, i.e., test suites with overall better values for the five objectives defined in Section 4.3. Using the Vargha and Delaney Statistics and the Mann-Whitney U test to analyze the results based on *HV*, we noticed that $SBI_{NSGA\text{-}II}$ performed significantly better than SBI_{RS} for both of the case studies, i.e., $\hat{A}_{12}$ for $SBI_{NSGA\text{-}II}$ is greater than 0.8 and the *p*-value is less than 0.05.

7.2.3 RQ2.3. Code Coverage

One can observe from Table D-5 that on average the test suites implanted by $SBI_{NSGA\text{-}II}$ have the highest code coverage. Specifically, the test suites implanted by $SBI_{NSGA\text{-}II}$ have 1) 4.2% and 6.5% higher *SC* and *BC* than SBI_{RS}, and 2) 4.5% and 6.9% higher *SC* and *BC* than SBI_{WGA}. Using the Vargha and Delaney Statistics and Mann-Whitney U test to analyze

the results, we observed that the test suites implanted by $SBI_{NSGA\text{-}II}$ had significantly higher SC and BC than both SBI_{RS} and SBI_{WGA} since $\hat{A}_{12}$ for both SC and BC are greater than 0.8 and the p-values are less than 0.001. Additionally, on comparing the test suites with similar estimated execution time (Table D-8), the implanted test suites using $SBI_{NSGA\text{-}II}$ achieved 1) 1.6% and 1.9% higher SC and BC than SBI_{RS}, and 2) 3.5% and 4.0% higher SC and BC than SBI_{WGA}. Therefore, we can conclude that $SBI_{NSGA\text{-}II}$ can achieve the highest code coverage.

7.2.4 RQ2.4. Mutation Score

From Table D-6, we can observe that the test suites implanted by $SBI_{NSGA\text{-}II}$ have the highest MS. Specifically, first, the average MS of the random test suites implanted by $SBI_{NSGA\text{-}II}$ have 1.37%, 1.16%, and 1.84% higher MS than the random test suites implanted by SBI_{RS}, selected test suites (based on equal importance to all the objectives) implanted by SBI_{RS}, and SBI_{WGA}. Second, the average MS of the selected test suites (based on equal importance to all the objectives) implanted by $SBI_{NSGA\text{-}II}$ have 2.38%, 2.17%, and 2.85% higher MS than the random test suites implanted by SBI_{RS}, selected test suites implanted by SBI_{RS}, and SBI_{WGA}. Moreover, while using the Vargha and Delaney Statistics and Mann-Whitney U test to analyze the results, we observed that the test suites implanted by $SBI_{NSGA\text{-}II}$ have significantly higher MS than both SBI_{RS} and SBI_{WGA} since $\hat{A}_{12}$ is greater than 0.8 and the p-value is less than 0.001. Thus, we can answer RQ2.4 as SBI can detect the highest fault (based on MS).

Notice that running time is an important perspective when evaluating a search-based approach [6, 63], and thus we report the running time of SBI. $SBI_{NSGA\text{-}II}$, SBI_{RS}, and SBI_{WGA} took an average of 48.1 minutes, 46.5 minutes, and 39.7 minutes for CS_1, and 64.6 minutes, 60.9 minutes, and 51.8 minutes for CS_2. Such a running time of the algorithm has no practical impact on the use of our approach since test case implantation is a one-time effort for a given test suite.

7.3 Overall Discussion

For RQ1.1 and RQ1.2, we observed that SBI managed to significantly increase the effectiveness of the original test suite (measured by $NCVV$ and $PCPV$) without significantly increasing the cost (measured by EET, NIT, and NCS). This is because SBI modifies the original test cases by changing (i.e., modifying/adding/removing) statements in test cases to maximize the effectiveness measures and minimize the cost measures as defined in Section 4.3.

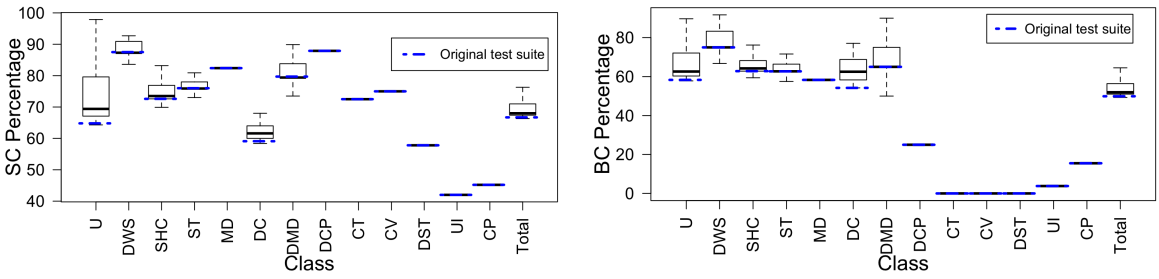

Fig. D-5. Average *SC* and *BC* for the Original Test Suite and Test Suites Implanted with SBI variants*

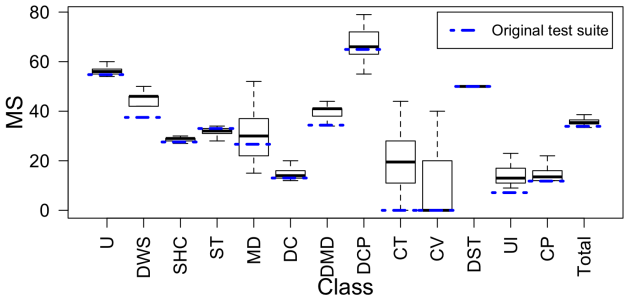

Fig. D-6. Average *MS* for the Original Solution compared with solutions from the three SBI variants*

*U: user, DWS: DeviceWindowSensor, SHC: SafeHomeConsole, ST: SensorTest, MD: MainDemo, DC: DeviceCamera, DMD: DeviceMotionDetector, DCP: DeviceControlPanelAbstract, CT: CameraTest, CV: CameraView, DST: DeviceSensorTester, UI: UserInterface, CP: ControlPanel.

Regarding RQ1.3, the results showed that SBI did not manage to improve *SC* and *BC* by a large percentage for CS_2. This is because SBI cannot further increase the code coverage if the original test suite has already covered all the parameters of a method in the source code or some methods are not targeted at all by the original test suite, which can be considered as the limitation of SBI and will be further investigated in the future. For instance, in the class *SensorTest* (available in [32]), all the parameters in method *armMotionDetector* have already been tested by the original test suite while method *actionPerformed* in the class was not targeted by the original test suite. Thus, SBI was not able to increase the code coverage for class *SensorTest*. Furthermore, Fig. D-5 presents *SC* and *BC* for the original test suite and the implanted test suites by SBI for 13 classes and the overall coverage (i.e., *Total*) that is the ratio of the total number of statements/branches covered and the total number of statements/branches present in the source code in CS_2. From Fig. D-5, we can observe that SBI managed to improve *SC* and *BC* for 6 of the 13 classes, and there was no change for the remaining 7 classes (e.g., *ControlPanel*) since all the parameters in the methods have already been tested or the original test suite does not target the methods.

Regarding RQ1.4, SBI increased *MS* for 5 classes (out of 13 classes) in CS_2 as shown in Fig. D-6. Note that *MS* did not increase in all the classes where *SC* and *BC* increased. For instance, *MS* increased in class *User* where *SC* and *BC* also grew. Class *DeviceCamera* [32] had an increasing *SC* and *BC*, but *MS* remained similar as compared with the original test suite. Such observation is consistent with the findings of the state-of-the-art [35] showing that the code coverage (e.g., *SC*) is not strongly correlated with test suite effectiveness (e.g., *MS*).

Regarding RQ2, $SBI_{NSGA\text{-}II}$ performed the best among the three SBI variants. This observation suggested that our test case implantation problem is not trivial to solve and requires an efficient optimization approach. $SBI_{NSGA\text{-}II}$ uses the *crossover* and *mutation*

operators to continuously evolve the original test suite as compared to SBI_{RS}, which does not use these two operators and only modifies the current test suite. Moreover, $SBI_{NSGA\text{-}II}$ produces a set of non-dominated solutions for preserving the optimal solutions with equivalent quality as compared to SBI_{WGA}, which might lose optimal solutions holding the same quality since it only stores one solution. Additionally, a weight-based genetic algorithm converts a multi-objective optimization problem into a single-objective problem by assigning weights to each objective, where weights play a primary role in the performance of the weight-based search algorithms. However, in practice, it is very difficult to set particular weights for different objectives.

Finally, for the real industrial case study, SBI needs to be run only once in practice (using 48.1 minutes for $SBI_{NSGA\text{-}II}$) to obtain an implanted test suite (including 118 test cases). This is equivalent to modifying one test case using on average 0.61 minutes (48.1/118) or 24.5 seconds. Clearly, modifying a test case manually within 24.5 seconds is practically impossible. In addition, SBI produces implanted test cases that satisfy various cost and effectiveness objectives (Section 4.3). Thus, we can conclude that SBI is beneficial in practice, at least for our industrial case study.

8 Threats to Validity

This section presents some of the potential threats to the validity of the two case studies investigated in this paper.

Threats to *internal validity* consider the internal factors (e.g., algorithm parameters) that could influence the obtained results [64]. In our context, we used the iRace optimization package [52] to tune the algorithm parameters, which has been widely used in the existing literature for automatic algorithm configuration [53-56]. Regarding the mutation rate applied on the test case level, we chose a rate that has earlier been investigated in the literature [27]. Another threat to *internal validity* involves instrumentation effects, i.e., the quality of the coverage information and mutation score measured [6]. To mitigate this threat, we used open source tools Eclemma and PIT that have been widely used in the literature [59, 60, 65, 66]. Regarding the *internal validity* threat concerning the implementation of the algorithms, we used all the algorithms implemented from the same tool (i.e., jMetal). Another *internal validity* threat arises regarding the suitability of the proposed five fitness functions. To address this issue, we will conduct additional experiments in the future to study the impact of the different fitness functions for configuration coverage.

Threats to *external validity* are related to the factors that affect the generalization of the results [64]. To mitigate this threat, we chose two different case studies (i.e., industrial case study and open source case study) for evaluating SBI. We plan to conduct more case studies in the future to generalize the results. It is also worth mentioning that such threats to *external validity* are common in empirical studies [67, 68]. Another *external validity* threat is due to the selection of search algorithms for SBI. To reduce this threat, we

selected the most widely used search algorithm (NSGA-II) that has been widely applied in different contexts [51, 67, 68], random search, and weight-based genetic algorithm.

Threats to *construct validity* exist when the comparison measurements are not comparable for all the algorithms [69]. To reduce *construct validity* threats, we used the same stopping criteria (i.e., 25,000 fitness evaluations) to find optimal solutions for all the algorithms across both of the case studies. Another threat to *construct validity* arises when the measurement metrics do not sufficiently cover the concepts they are supposed to measure [6]. To mitigate this threat, we compare the implanted test suites by SBI and the original test suite based on evaluation metrics that have been widely adopted in the literature: statement coverage, branch coverage, mutation score, and running time.

Threats to *conclusion validity* are related to the factors that influence the conclusion drawn from the results of the experiments [70]. The *conclusion validity* threat when using randomized algorithms is related to random variation in the produced results. To mitigate this threat, we repeated each experiment 10 times for each algorithm in SBI to reduce the possibility that the results were obtained by chance. Moreover, we carefully applied statistical tests by following the guidelines for reporting results for randomized algorithms [45].

9 Related Work

There are a number of existing works related to code transplantation, test suite augmentation, test generation, and testing of highly configurable software systems that have certain similarities with our work (i.e., test case implantation). We discuss each of them in detail as below.

9.1 Code Transplantation

In recent years, there has been an increasing attention on code transplantation within/across software systems [71-74]. For instance, Weimer et al. [71] used genetic programming (GP) to evolve defective programs to fix defects while maintaining specified functionalities for automatic program repair. Petke et al. [73] used GP to evolve a program by transplanting code from other programs for improving the system's performance. Barr et al. [72] automatically transplanted functionalities of programs across different software systems using GP and program slicing.

As compared with the existing work for code transplantation (e.g., [71-73]), SBI has at least two key differences: 1) The goal is different, i.e., we aim at automatically implanting existing test cases to test untested configurations rather than transplanting software code; 2) Five objectives (e.g., maximizing the number of configuration variable values covered) are defined to guide the search for selecting and implanting test cases.

9.2 Test Suite Augmentation

There is a number of studies focusing on test suite augmentation that refers to identifying code elements affected by software changes as it evolves (e.g., new functionalities are added), and generating test cases to test those elements [75-79]. For instance, dependence analysis and partial symbolic execution were used in [77] to identify the changed test requirements when the program is evolved, however, they do not generate test cases. In [75] a directed test suite augmentation technique was proposed for 1) identifying the code affected by changes in the program and 2) generating new test cases for testing the affected code using a concolic test case generation approach [80].

As compared with the above-mentioned literature, SBI aims to cost-effectively increase the configuration coverage of the original test suite rather than generating new test cases for testing the modified code. Furthermore, we defined three operations (i.e., *addition*, *modification* and *deletion*) to automatically implant the test cases, which is not the case in the existing work.

9.3 Test Generation

Different techniques have been used for test generation such as random testing [81], symbolic execution [82, 83] and search techniques [27, 31, 84-86] (that is the most relevant to this work). For instance, Miller et al. [85] used program dependence graphs and a genetic algorithm to generate test data for maximizing condition-decision coverage. Ali et al. [84] designed a search-based Object Constraint Language (OCL) constraint solver by defining branch distance functions to support test data generation for model-based testing. Fraser and Arcuri [27, 31] designed and implemented a tool (i.e., EvoSuite) to generate test cases with an aim to maximize different coverage criteria (e.g., line, branch) and mutation testing using search. As compared with the state-of-the-art for test generation, SBI focuses on automated implanting an existing test suite to test untested configurations instead of generating test cases from scratch.

9.4 Testing of Highly Configurable Software Systems

There is a large body of research with respect to the testing of highly configurable software systems with many configurations [87-93]. Existing works have proposed many sampling techniques to select a subset of representative configurations for testing [90-93]. For instance, Swanson [90] modeled a highly configurable system using a feature model followed by applying random sampling to repeatedly generate a random configuration from the feature model for testing. Qu et al. [91] and Yilmaz et al. [92] used covering array sampling method to generate at least one *t*-combination configuration (to be tested) for representing all valid *t*-combination configurations in the configuration space. Cohen et al. [94] combined pairwise algorithms (e.g., metaheuristic search algorithm) with Boolean

satisfiability (SAT) solvers to handle constraints while generating configurations for interaction testing of highly configurable systems.

As compared with the existing studies of testing highly configurable software systems, the focus of our work is totally different, i.e., we aim at implanting an original test suite to test untested configurations, and thus increase the configuration coverage of the existing test suite. To achieve this goal, we proposed a search-based approach (i.e., SBI) for automated test case implantation (Section 5).

9.5 Multi-Objective Regression Test Optimization

Multi-objective regression test optimization aims to select a subset of test cases that gives the maximum cost-value benefit and ordering test cases such that early attainment of cost-value tradeoff is achieved [95]. There exists a large body of research for multi-objective regression test optimization, e.g., [6, 96-98]. While those work focus on selecting/minimizing/prioritizing test cases based on particular objectives (e.g., maximizing code coverage) without changing the original test suite, SBI implants the original test suite to test untested configurations by modifying the test cases in a cost-effective manner. After the original test suite has been implanted using SBI, the modified test cases can be prioritized/selected/minimized based on defined objectives for regression testing as discussed in some of our prior work [5, 62, 99, 100].

10 Conclusion

This paper introduced a novel search-based test case implantation approach (SBI) including two key components (i.e., *test case analyzer* and *test case implanter*) with the aim to automatically analyze and implant the existing test cases to test the untested configurations. Three variants of SBI (with NSGA-II, weight-based GA, and RS) were evaluated using one industrial and one open source case studies. The results showed that the test suites implanted by all the three SBI variants performed significantly better than the original test suite for both the case studies. Additionally, SBI with NSGA-II performed the best of the three. Specifically, SBI significantly outperformed the original suite for both the case studies by achieving on average 19.3% higher number of configuration variables values and 57.0% higher pairwise coverage of parameter values of test API commands. Moreover, for the open source case study, the implanted test suites managed to improve statement coverage, branch coverage, and mutation score with on average 5.0%, 7.9%, and 3.2%, respectively.

As a future work, we plan to conduct additional experiments to study the impact of the proposed fitness functions in configuration coverage, statement coverage, branch coverage, and mutation score. Moreover, we plan to apply more case studies to further strengthen the applicability of SBI.

Acknowledgement

This research was supported by the Research Council of Norway (RCN) funded Certus SFI. Shuai Wang is also supported by RFF Hovedstaden funded MBE-CR project and COST Action CA15140 (ImAppNIO). Tao Yue and Shaukat Ali are supported by RCN funded Zen-Configurator project, RCN funded MBTCPS project, and COST Action IC1404.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*: John Wiley & Sons, 2011.

[2] S. A. Asadollah, R. Inam, and H. Hansson, "A Survey on Testing for Cyber Physical System," in *IFIP International Conference on Testing Software and Systems*, pp. 194-207. Springer, 2015.

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* vol. 6, no. 2, pp. 182-197. IEEE, 2002.

[4] *SafeHome Project*. Available: https://github.com/Suckzoo/CS350

[5] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "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*, pp. 182-191. ACM, 2016.

[6] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering,* vol. 33, no. 4, pp. 225-237. IEEE, 2007.

[7] A. S. Sayyad and H. Ammar, "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*, pp. 21-27. IEEE, 2013.

[8] J. Brownlee, *Clever algorithms: nature-inspired programming recipes*: Lulu, 2011.

[9] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety,* vol. 91, no. 9, pp. 992-1007. Elsevier, 2006.

[10] *Unit testing framework*. Available: https://docs.python.org/2/library/unittest.html

[11] B. Korel and A. M. Al-Yami, "Automated regression test generation," *ACM SIGSOFT Software Engineering Notes,* vol. 23, no. 2, pp. 143-152. 1998.

[12] R. Pandita, T. Xie, N. Tillmann, and J. De Halleux, "Guided test generation for coverage criteria," in *IEEE International Conference on Software Maintenance (ICSM), 2010*, pp. 1-10. IEEE, 2010.

[13] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting, "Boundary coverage criteria for test generation from formal models," in *15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2004.* , pp. 139-150. IEEE, 2004.

[14] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, pp. 419-430. 2006.

[15] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR),* vol. 43, no. 2, p. 11. 2011.

[16] W. D. Blizard, "Multiset theory," *Notre Dame Journal of formal logic,* vol. 30, no. 1, pp. 36-66. 1988.

[17] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial software testing," *Computer,* vol. 42, no. 8, 2009.

[18] Y. Lei and K.-C. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *Proceedings of the Third IEEE International Symposium on High-Assurance Systems Engineering*, pp. 254-261. IEEE, 1998.

[19] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Transactions on Software Engineering,* vol. 40, no. 11, pp. 1100-1125. 2014.

[20] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology, 2007. QUATIC 2007.*, pp. 30-39. IEEE, 2007.

[21] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *ACM Sigplan Notices*, pp. 177-184. ACM, 1984.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 9, no. 3, pp. 319-349. 1987.

[23] J. Zhao, "Applying program dependence analysis to Java software," in *Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pp. 162-169. 1998.

[24] S. Ali and T. Yue, "Evaluating Normalization Functions with Search Algorithms for Solving OCL Constraints," in *Testing Software and Systems*, ed: Springer, 2014, pp. 17-31.

[25] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International Static Analysis Symposium*, pp. 40-56. Springer, 2001.

[26] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, pp. 439-449. IEEE Press, 1981.

[27] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering,* vol. 39, no. 2, pp. 276-291. 2013.

[28] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proceedings of 11th International Conference on Quality Software (QSIC), 2011*, pp. 31-40. IEEE, 2011.

[29] R. S. Pressman, *Software engineering: a practitioner's approach*: Palgrave Macmillan, 2005.

[30] D. Wheeler. *Sloccount*. Available: https://www.dwheeler.com/sloccount/

[31] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416-419. ACM, 2011.

[32] *Supplimentary material*. Available: https://sbi.netlify.com/

[33] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 179-188. IEEE, 1999.

[34] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th international conference on Software Engineering*, pp. 100-107. IEEE Computer Society Press, 1993.

[35] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 435-445. ACM, 2014.

[36] J. S. Kracht, J. Z. Petrovic, and K. R. Walcott-Justice, "Empirically evaluating the quality of automatically generated and manually written test suites," in *Proceedings of the 14th International Conference on Quality Software (QSIC), 2014* pp. 256-265. IEEE, 2014.

[37] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 654-665. ACM, 2014.

[38] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 72-82. ACM, 2014.

[39] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach," *IEEE Transactions on Evolutionary Computation,* vol. 3, no. 4, pp. 257-271. IEEE, 1999.

[40] J. Knowles, L. Thiele, and E. Zitzler, "A tutorial on the performance assessment of stochastic multiobjective optimizers," *Tik report,* vol. 214, pp. 327-332. ETH Zurich, 2006.

[41] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "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*, pp. 631-642. ACM, 2016.

[42] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham, "AbYSS: Adapting scatter search to multiobjective optimization," *IEEE Transactions on Evolutionary Computation,* vol. 12, no. 4, pp. 439-457. IEEE, 2008.

[43] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika,* vol. 52, no. 3-4, pp. 591-611. 1965.

[44] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*: crc Press, 2003.

[45] A. Arcuri and L. Briand, "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)*, pp. 1-10. IEEE, 2011.

[46] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132. 2000.

[47] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics,* pp. 50-60. JSTOR, 1947.

[48] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software,* vol. 42, no. 10, pp. 760-771. Elsevier, 2011.

[49] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. C. Coello, F. Luna, and E. Alba, "Smpso: A new pso-based metaheuristic for multi-objective optimization," in *Proceedings of the Symposium on Computational Intelligence in Multi-criteria Decision-making*, pp. 66-73. IEEE, 2009.

[50] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba, "Solving three-objective optimization problems using a new hybrid cellular genetic algorithm," in *Parallel Problem Solving from Nature–PPSN X*, ed: Springer, 2008, pp. 661-670.

[51] A. S. Sayyad, T. Menzies, and H. Ammar, "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*, pp. 492-501. IEEE, 2013.

[52] M. López-Ibánez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package, iterated race for automatic algorithm configuration," Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium2011.

[53] T. Liao, M. A. M. de Oca, and T. Stützle, "Computational results for an automatically tuned CMA-ES with increasing population size on the CEC'05 benchmark set," *Soft Computing,* vol. 17, no. 6, pp. 1031-1046. 2013.

[54] L. P. Cáceres, M. López-Ibáñez, and T. Stützle, "Ant colony optimization on a limited budget of evaluations," *Swarm Intelligence,* vol. 9, no. 2-3, pp. 103-124. 2015.

[55] Z. Ren, H. Jiang, J. Xuan, S. Zhang, and Z. Luo, "Feature based problem hardness understanding for requirements engineering," *Science China Information Sciences,* vol. 60, no. 3, p. 032105. 2017.

[56] L. Bezerra, M. López-Ibánez, and T. Stützle, "Automatic Configuration of Multi-objective Optimizers and Multi-objective Configuration," Technical Report TR/IRIDIA/2017-011, IRIDIA, Université Libre de Bruxelles, Belgium2017.

[57] M. R. Hoffmann, J. Brock, and E. Mandrikov. *Java Code Coverage for Eclipse*. Available: http://www.eclemma.org/

[58] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449-452. ACM, 2016.

[59] L. Inozemtseva, H. Hemmati, and R. Holmes, "Using fault history to improve mutation reduction," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pp. 639-642. ACM, 2013.

[60] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, *et al*., "Predictive mutation testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 342-353. ACM, 2016.

[61] M. Zhang, S. Ali, T. Yue, and M. Hedman, "Uncertainty-based Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach," *Simula Research Laboratory,* 2016.

[62] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software,* vol. 103, pp. 370-391. Elsevier, 2015.

[63] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software,* vol. 83, no. 4, pp. 689-701. Elsevier, 2010.

[64] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*: John Wiley & Sons, 2012.

[65] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 1145-1156. ACM, 2016.

[66] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, *et al*., "A cross-tool communication study on program analysis tool notifications," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 73-84. ACM, 2016.

[67] S. Wang, S. Ali, T. Yue, and M. Liaaen, "UPMOA: An improved search algorithm to support user-preference multi-objective optimization," in *Proceedings of the 26th International Symposium on Software Reliability Engineering*, pp. 393-404. IEEE, 2015.

[68] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 619-630. ACM, 2016.

[69] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Software,* vol. 12, no. 4, p. 52. IEEE, 1995.

[70] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, "Experimentation in software engineering: an introduction. 2000," ed: Kluwer Academic Publishers, 2000.

[71] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 364-374. IEEE Computer Society, 2009.

[72] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 257-269. ACM, 2015.

[73] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *European Conference on Genetic Programming*, pp. 137-149. Springer, 2014.

[74] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard, "Automatic error elimination by multi-application code transfer," 2014.

[75] Z. Xu and G. Rothermel, "Directed test suite augmentation," in *Asia-Pacific Software Engineering Conference, 2009. APSEC'09.* , pp. 406-413. IEEE, 2009.

[76] S. Yoo and M. Harman, "Test data augmentation: generating new test data from existing test data," *Centre for Research on Evolution, Search & Testing (CREST),* 2008.

[77] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 218-227. IEEE Computer Society, 2008.

[78] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 257-266. ACM, 2010.

[79] Z. Xu, M. B. Cohen, W. Motycka, and G. Rothermel, "Continuous test suite augmentation in software product lines," in *Proceedings of the 17th International Software Product Line Conference*, pp. 52-61. ACM, 2013.

[80] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ACM SIGSOFT Software Engineering Notes*, pp. 263-272. ACM, 2005.

[81] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software: Practice and Experience,* vol. 34, no. 11, pp. 1025-1050. 2004.

[82] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *International*

Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 365-381. Springer, 2005.

[83] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*, pp. 134-153. Springer, 2008.

[84] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A search-based OCL constraint solver for model-based test data generation," in *11th International Conference on Quality Software (QSIC), 2011*, pp. 41-50. IEEE, 2011.

[85] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Information and Software Technology,* vol. 48, no. 7, pp. 586-605. 2006.

[86] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1098-1105. ACM, 2007.

[87] M. Lochau, S. Oster, U. Goltz, and A. Schürr, "Model-based pairwise testing for feature interaction coverage in software product line engineering," *Software Quality Journal,* vol. 20, no. 3-4, pp. 567-604. 2012.

[88] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 445-454. ACM, 2010.

[89] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering,* vol. 30, no. 6, pp. 418-421. 2004.

[90] J. Swanson, "A Self-Adaptive Framework for Failure Avoidance in Configurable Software," 2014.

[91] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp. 75-86. ACM, 2008.

[92] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering,* vol. 32, no. 1, pp. 20-34. 2006.

[93] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Pairwise testing for software product lines: comparison of two approaches," *Software Quality Journal,* vol. 20, no. 3-4, pp. 605-643. 2012.

[94] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 129-139. ACM, 2007.

[95] M. Harman, "Making the case for MORTO: Multi objective regression test optimization," in *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 111-114. IEEE, 2011.

[96] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability,* vol. 22, no. 2, pp. 67-120. Wiley Online Library, 2012.

[97] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed, "Multi-objective test case prioritization in highly configurable systems: A case study," *Journal of Systems and Software,* vol. 122, pp. 287-310. 2016.

[98] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 140-150. ACM, 2007.

[99] D. Pradhan, S. Wang, S. Ali, and T. Yue, "Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1085-1092. ACM, 2016.

[100] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "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*, pp. 172-190. Springer, 2016.

Paper E

Employing Rule Mining and Multi-Objective Search for Dynamic Test Case Prioritization

Dipesh Pradhan, Shuai Wang, Tao Yue,
Shaukat Ali, Marius Liaaen

Submitted to Journal of Systems
and Software, May 2018.

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 sets 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.

Keywords: Multi-objective Optimization; Rule Mining; Dynamic Test case Prioritization; Search; Black-box Regression Testing.

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 [1-3]. 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 [4, 5], 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 [6-9]. 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 [7, 10-14], i.e.,

they produce a list of static prioritized test cases. Based on our collaboration with Cisco Systems [15, 16] 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 minutes) *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 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) [17] 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) [18]). 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) [19]) 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 [20], and c) Pruning Rule-Based Classification (PART) [21] together with 2) three different search algorithms: a) NSGA-II, b) Strength Pareto Evolutionary Algorithm (SPEA2) [22], and c) Indicator-based Evolutionary Algorithm (IBEA) [23] for a total of nine different configurations of REMAP (i.e., three rule mining algorithms × 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 [24] and IOF/ROL [24], and 3) one open Google Shared Dataset of Test Suite Results (GSDTSR) [25]. We compared the 18 configurations of REMAP with 1) two variants of random search (RS) [10, 12, 26]: RS_{2obj} based on FDC and TRS, and RS_{3obj} based on FDC, TRS, and EET, 2) three prioritization approaches based on Greedy [6, 8, 27]: 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 [28-30]: nine each with two objectives and three objectives, and 3) six variants of rule-based TP (RBP) approaches [31] (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 [17] 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 background section has been added to introduce data mining and multi-objective optimization.
4) 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.
5) A more in-depth discussion has been added based on the results.

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 [32]. This process is achieved using pattern recognition technologies along with statistical and mathematical techniques [33]. 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 [33].

Since all automated systems generate some form of data for diagnostic or analysis objectives, a large amount of data are produced [34]. 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 [34]. 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) [35]. On the other hand, unsupervised learning aims to identify hidden patterns inside input data without labeled responses. More specifically, unsupervised learning group instances without a pre-specified dependent attribute [35]. Additionally, supervised learning uses class information of the training instances while unsupervised learning does not use the class information. In our context, we aim to find the hidden rules between the execution relations of the test cases such that each test case execution result (e.g., *pass*) is used as a class and the past execution history is used as the training instances. Therefore, 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 regressor maps the input space into a real-valued domain [35]. Since we have predefined classes (i.e., test case execution results), we use classification models. The classification rules are constructed in two major ways: 1) Indirect method (e.g., C4.5 [20]), which learns decision trees then converts them to rules and 2) Direct method (e.g., Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [18]), which extracts rules directly from the data. 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. On the other hand, 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 [18]. 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 [36]. The Pruning Rule-Based Classification (PART) [21] is derived from C4.5 and RIPPER to avoid their shortcomings. More specifically, PART creates partial trees and corresponding to each partial tree, a single rule is extracted for the branch that covers the maximum nodes [36].

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 is called as Pareto fronts [37-39]. *Pareto optimality* defines the Pareto dominance to assess the quality of solutions [40]. Suppose a multi-objective TP problem consists of m objectives to be optimized, $O = \{o_1, o_2, \dots, o_m\}$, and each objective can be measured using a fitness f_i from $F = \{f_1, f_2, \dots, f_m\}$. 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*:

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

Multi-objective search algorithms have been widely applied for different multi-objective test prioritization problems. We employed three representative multi-objective search algorithms from the literature [41]: Non-dominated Sorting Genetic Algorithm II (NSGA-II) [19], Strength Pareto Evolutionary Algorithm (SPEA2) [22], and Indicator-based Evolutionary Algorithm (IBEA) [23].

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 [19]. 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 [39].

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) [22]. 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 [23]. Specifically, *HV* calculates the volume in the objective space covered by a non-dominated set of solutions (e.g., Pareto front) [42]. 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 [8]. 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 [43] 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) [15, 16, 29]. 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 minutes). 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 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 [9, 24, 31].

Fig. E-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. E-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.

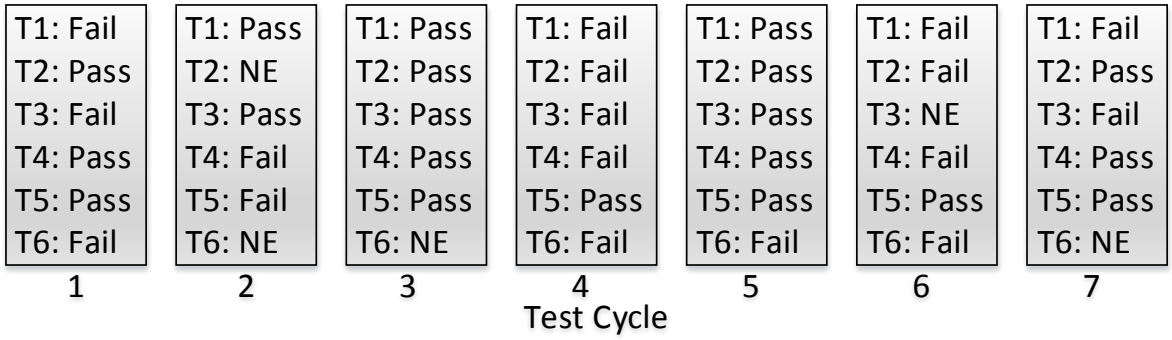

Fig. E-1. Sample data showing the execution history for six test cases*

*NE: Not 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. E-1), the corresponding *fail* test cases (e.g., T_3 in Fig. E-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. E-1), the priority of the related *pass* test case(s) (T_2 in Fig. E-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., $TS = \{T_1, T_2, \dots, T_n\}$. For instance, in the running example (Fig. E-1), TS consists of six test cases.

Notation 2. TC is a set of p test cycles that have been executed, i.e., $TC = \{tc_1, tc_2, \dots, tc_p\}$, such that one or more test cases are executed in each test cycle. For example, there are seven test cycles in the example (Fig. E-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. E-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 , 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, $S = \{s_1, s_2, \dots, s_q\}$, where $q = n!$. For instance in the running example with six test cases (Fig. E-1), the total number of solutions $q = 720$. Each solution s_j is an order of test cases from TS, $s_j = \{T_{j1}, T_{j2}, \dots, T_{jn}\}$ 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 $s_f = \{T_{f1}, T_{f2}, \dots, T_{fn}\}$ 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 $s_f = \{T_{f1}, T_{f2}, \dots, T_{fn}\}$, the dynamic TP problem aims to update the execution order of the test cases in s_f to obtain a solution $s_f{}' = \{T_{f1}{}', \dots, T_{fn}{}'\}$, 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 (Section 5.2 – Section 5.4).

5.1 Overview of REMAP

Fig. E-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. E-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 input 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. E-2) to detect the fault(s) as soon as possible.

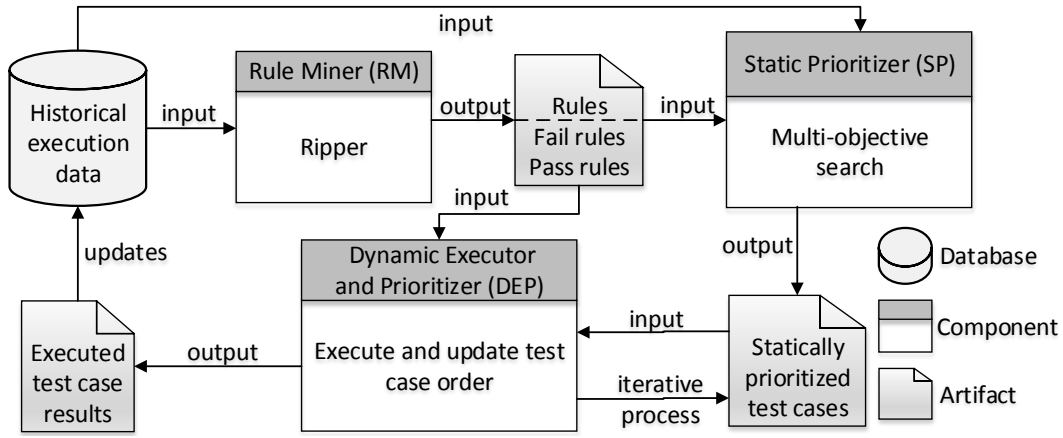

Fig. E-2. Overview of REMAP

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: $(V(T_i)\ AND, \dots, AND\ V(T_k)) \xrightarrow{fail} \big(V(T_j) = fail\big) \wedge T_j \notin \{T_i, \dots, T_k\}$, 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) as is often used in literature [44-47]. For example, in Fig. E-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. E-1). With *fail rules*, we aim to prioritize and execute the test cases that are 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: $(V(T_i)\ AND, \dots, AND\ V(T_k)) \xrightarrow{pass} \big(V(T_j) = pass\big) \wedge T_j \notin \{T_i, \dots, T_k\}$ 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) [44-47]. For instance, in Fig. E-1, there exists a *pass rule* between T_2 and T_4: $(V(T_4) = pass) \xrightarrow{pass} (V(T_2) = pass)$, 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. E-1). With *pass rules*, we aim to deprioritize 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*.

Input: Set of test cycles $TC = \{tc_1, tc_2, \dots, tc_p\}$, set of test cases $TS = \{T_1, T_2, \dots, T_n\}$
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. E-3. Overview of rule miner

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. E-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. E-3). Afterward, the mined rules are classified into a *fail rule* or *pass rule* one at a time (lines 4-8 in Fig. E-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. E-3); otherwise, the rule is classified as a *pass rule* (lines 7 and 8 in Fig. E-3). For instance, in Fig. E-1, two rules are mined for T_1 using RIPPER: $(V(T_3) = fail) \xrightarrow{fail} (V(T_1) = fail)$ as a *fail rule* and $(V(T_3) = pass) \xrightarrow{pass} (V(T_1) = pass)$ as a *pass rule*. This process of rule mining and classification is repeated for all the test cases in the test suite (lines 1-8 in Fig. E-3), and finally, the mined *fail rule* and *pass rule* are returned (line 9 in Fig. E-3).

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

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

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

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 [12, 13, 24, 48, 49]. For example, T_1 and T_6 failed the highest number of times (i.e., four times) in Fig. E-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. E-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 E-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. E-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 [13, 29]. 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. E-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. E-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. E-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 E-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. E-1, if the execution time of T_1 for seven executions in the seven test cycles is 10.2 minutes (m), 10.8m, 10.4m, 9.6m, 9.9m, 10.3m, 10.6m; then the EET of T_1 is 10.3m. 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 FDC 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 [37-39] 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 [50]. 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. E-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. E-4, and it is explained using the example (Section 3) in Fig. E-5.

Let us assume a prioritization solution is obtained as $\{T_1, T_4, T_3, T_6, T_2, T_5\}$ (Fig. E-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 E-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. E-4). For example in Fig. E-1, T_6 always had the verdict *fail* in historical execution data, and thus T_6 is added to AF.

Input: Solution $s_f = \{T_{f1}, \dots, T_{fn}\}$, *pass rules* PR, *fail rules* FR
Output: Dynamically prioritized test cases
Begin:

```
1  k ⟵ 1
2  AF ⟵ AlwaysFailing(s_f)                    // set of test cases that always fail
3  AP ⟵ AlwaysPassing(s_f)                    // set of test cases that always pass
4  while (k ≤ n) and (termination_conditions_not_satisfied) do
5     TE ⟵ Get − test − case − execution(s_f, AF, AP)
6     verdict ⟵ execute(TE)                  // execute the test case
7     if (verdict = 'pass') then
8        if (TE ∈ 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 ∈ FR) then                   // if the test case has a fail rule
12          move the related test cases forward to execute next
13    s_final ⟵ s_final ∪ TE                 // dynamically prioritized solution
14    k ⟵ k + 1
15    remove TE from s_f                     // remove executed test case from s_f
16 return s_final
End
```

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

Afterward, *DEP* identifies the test cases to execute using the algorithm *Get-test-case-execution* (Algorithm E-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 E-1), it is selected for execution (line 2 in Algorithm E-1), removed from AF (line 3 in Algorithm E-1), and returned (line 12 in Algorithm E-1) to *DEP* (Fig. E-4). For instance, initially, T_6 is selected for execution from AF in Fig. E-5. Afterward, the selected test case (i.e., T_6) is executed (line 6 in Fig. E-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. E-4). Then, the executed test case is added to the final solution (line 13 in Fig. E-4) and removed from the statically prioritized solution for execution (line 15 in Fig. E-4). If there exists no related test case(s), the next test case is selected for execution (line 5 in Fig. E-4).

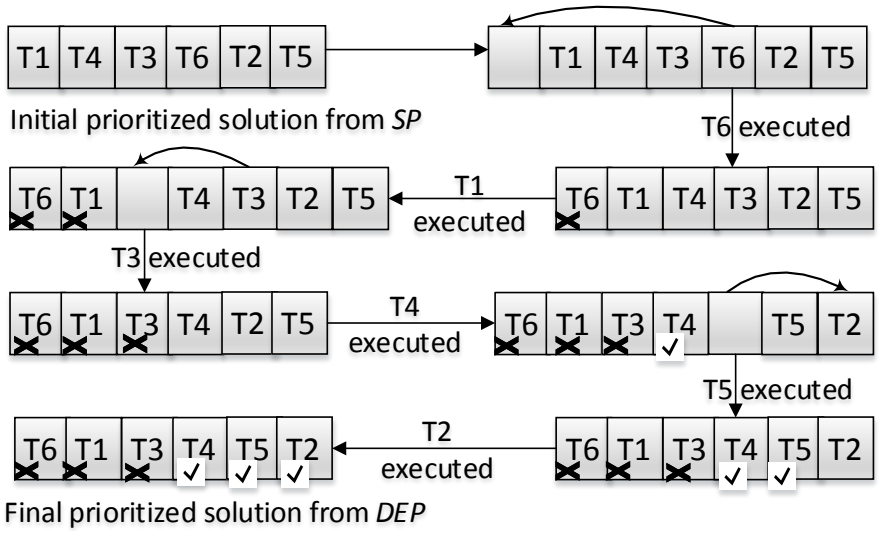

Fig. E-5. Example of dynamic test case execution and prioritization

Algorithm G-1: Get-test-case-execution

Input: Solution $s_f = \{T_{f1}, \dots, T_{fn}\}$, 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 ← F(AF_0)                    // get the first fail test case
3     remove AF_0 from AF             // remove the fail test case
4   else TE ← F(s_f0)                 // if there are not any test cases in AF
5     move_T ← True                   // initially test cases can be moved
6     first_T ← 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 ← F(s_f0)
10      if (first_T = TE) then
11        move_T ← False              // do not move test case anymore
12  return TE
End
```

To select the next test cases, Algorithm E-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 Algorithm E-1). 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 E-1). For example, T_1 is executed after T_6 in Fig. E-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. E-4), *pass rule* or *fail rule* is employed once more (lines 7-12 in Fig. E-4). If the test case has the verdict *fail* (line 10 in Fig. E-4), it is checked if it is a part of any *fail rule* (line 11 in Fig. E-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. E-4), e.g., T_3 is moved in front of T_4 in Fig. E-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. E-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. E-5 since T_4 *passed* and there is a *pass rule* between T_4 and T_2 as shown in Table E-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. E-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. E-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. E-2. Note that REMAP updates the mined rules after each test cycle rather than after executing each test case since the mined rules will not be changed largely if updated after executing each test case and rule mining is computationally expensive [51, 52].

Note that REMAP includes *Static Prioritizer* (i.e., *SP* in Section 5.3) before *Dynamic Exe cutor 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 E-2) with research questions (Section 6.1), case studies (Section 6.2), experiment tasks and evaluation metric (Section 6.3), and statistical tests and experiment settings (Section 6.4).

Table E-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	$BestREMAP_{3obj}$, $BestRBP_{2obj}$, $BestRBP_{3obj}$				

6.1 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) $\text{REMAP}_{\text{C4.5+IBEA(2obj)}}$, 2) $\text{REMAP}_{\text{C4.5+NSGA-II(2obj)}}$, 3) $\text{REMAP}_{\text{C4.5+SPEA2(2obj)}}$, 4) $\text{REMAP}_{\text{RIPPER+IBEA(2obj)}}$, 5) $\text{REMAP}_{\text{RIPPER+NSGA-II(2obj)}}$, 6) $\text{REMAP}_{\text{RIPPER+SPEA2(2obj)}}$, 7) $\text{REMAP}_{\text{PART+IBEA(2obj)}}$, 8) $\text{REMAP}_{\text{PART+NSGA-II(2obj)}}$, 9) $\text{REMAP}_{\text{PART+SPEA2(2obj)}}$, 10) $\text{REMAP}_{\text{C4.5+IBEA(3obj)}}$, 11) $\text{REMAP}_{\text{C4.5+NSGA-II(3obj)}}$, 12) $\text{REMAP}_{\text{C4.5+SPEA2(3obj)}}$, 13) $\text{REMAP}_{\text{RIPPER+IBEA(3obj)}}$, 14) $\text{REMAP}_{\text{RIPPER+NSGA-II(3obj)}}$, 15) $\text{REMAP}_{\text{RIPPER+SPEA2(3obj)}}$, 16) $\text{REMAP}_{\text{PART+IBEA(3obj)}}$, 17) $\text{REMAP}_{\text{PART+NSGA-II(3obj)}}$, 18) $\text{REMAP}_{\text{PART+SPEA2(3obj)}}$. Moreover, we used RS with two objectives (i.e., RS_{2obj}) and three objectives (i.e., 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) $\text{SSBP}_{\text{C4.5+IBEA(2obj)}}$, 2) $\text{SSBP}_{\text{C4.5+NSGA-II(2obj)}}$, 3) $\text{SSBP}_{\text{C4.5+SPEA2(2obj)}}$, 4) $\text{SSBP}_{\text{RIPPER+IBEA(2obj)}}$, 5) $\text{SSBP}_{\text{RIPPER+NSGA-II(2obj)}}$, 6) $\text{SSBP}_{\text{RIPPER+SPEA2(2obj)}}$,

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 [31], 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) $RBP_{C4.5\text{-}2obj}$, 2) $RBP_{RIPPER\text{-}2obj}$, 3) $RBP_{PART\text{-}2obj}$, 4) $RBP_{C4.5\text{-}3obj}$, 5) $RBP_{RIPPER\text{-}3obj}$, and 6) $RBP_{PART\text{-}3obj}$. 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.

6.2 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) [24] and IOF/ROL (CS_4) [24], and Google Shared Dataset of Test Suite Results (GSDTSR) (CS_5) [25]. For each case study, test case execution result is linked to a particular test cycle such that each test cycle is considered as an occurrence of regression testing. More specifically, each case study contains historical execution data of the test cases for more than 300 test cycles as shown in Table E-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* for each test case (Section 5.3). Note that the open source case studies for CS_3 - CS_5 are publicly available at [53].

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

Case Study	Data Set	#Test Cases	#Test Cycles	#Verdicts
CS_1	Cisco Data Set1	60	8,322	296,042
CS_2	Cisco Data Set2	624	6,302	149,039
CS_3	ABB Paint Control	89	351	25,497
CS_4	ABB IOF/ROL	1,941	315	32,162
CS_5	Google GSDTSR	5,555	335	1,253,464

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 E-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 E-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 E-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 6,303 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 [53], 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 [24, 31].

6.3.2 Evaluation Metric

We used Average Percentage of Fault Detected per Cost metric ($APFD_c$) proposed by Elbaum et al. [26] 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 [7] 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 [54-56]. The $APFD_c$ for a solution can be

calculated as: $APFD_c = \frac{\sum_{i=1}^{m}\left(\sum_{j=TF_i}^{n} et_j - \frac{1}{2}et_{TF_i}\right)}{n \times m}$, 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 [57], the Vargha and Delaney $\hat{A}_{12}$ statistics [58], Mann-Whitney U test [59], Kruskal-Wallis test [60], and Dunn's test [61] with Bonferroni Correction [62] are used to statistically evaluate the results for the five research questions as shown in Table E-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 [63] 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.,${}^{9}C_2$, 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.

6.4.2 Experiment Settings

We used the WEKA data mining software [64] 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. We employed a widely-used Java framework jMetal [65] to implement the component *SP* (Section 5.3), and the standard settings were applied to configure the search algorithms (e.g., NSGA-II) as are usually recommended [57]. More specifically, the population size is set as 100, the crossover rate is 0.9, the mutation rate is 1/(Total number of test cases), and the maximum number of fitness evaluation (i.e., termination criteria of the algorithm) is set

as 50,000. 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 [66].

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 [67].

Moreover, Table E-4 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 [67]. Fig. E-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}.

Table E-4. $APFD_c$ scores in percentage for different approaches for the 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						

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 E-5. Ranking of 18 variants of REMAP with two objectives and three objectives for the five case studies*

CS	# of obj	Rank								
		1	2	3	4	5	6	7	8	9
CS_1	2	$REMAP_6$/$REMAP_4$		$REMAP_5$	$REMAP_1$/$REMAP_2$		$REMAP_3$	$REMAP_8$	$REMAP_7$/$REMAP_9$	
	3	$REMAP_4$	$REMAP_5$	$REMAP_6$	$REMAP_1$	$REMAP_2$	$REMAP_3$	$REMAP_8$	$REMAP_9$	$REMAP_7$
CS_2	2	$REMAP_7$	$REMAP_9$	$REMAP_4$	$REMAP_6$/$REMAP_8$		$REMAP_1$/$REMAP_5$		$REMAP_2$/$REMAP_3$	
	3	$REMAP_4$	$REMAP_7$	$REMAP_1$	$REMAP_9$	$REMAP_3$	$REMAP_2$	$REMAP_5$	$REMAP_6$	$REMAP_8$
CS_3	2	$REMAP_6$	$REMAP_5$	$REMAP_4$	$REMAP_3$	$REMAP_9$	$REMAP_1$/$REMAP_7$		$REMAP_2$/$REMAP_8$	
	3	$REMAP_4$	$REMAP_7$	$REMAP_1$	$REMAP_6$	$REMAP_3$	$REMAP_9$	$REMAP_5$	$REMAP_2$	$REMAP_8$
CS_4	2	$REMAP_1$	$REMAP_3$	$REMAP_2$	$REMAP_5$/$REMAP_6$/$REMAP_7$/$REMAP_8$/$REMAP_9$					$REMAP_4$
	3	$REMAP_7$	$REMAP_3$	$REMAP_9$	$REMAP_1$	$REMAP_2$	$REMAP_4$	$REMAP_6$	$REMAP_5$	$REMAP_8$
CS_5	2	$REMAP_5$	$REMAP_3$	$REMAP_2$	$REMAP_1$	$REMAP_4$/$REMAP_6$/$REMAP_7$/$REMAP_8$/$REMAP_9$				
	3	$REMAP_9$	$REMAP_4$	$REMAP_6$	$REMAP_1$	$REMAP_3$	$REMAP_7$	$REMAP_8$	$REMAP_2$	$REMAP_5$

*$REMAP_1$: $REMAP_{C4.5+IBEA}$, $REMAP_2$: $REMAP_{C4.5+NSGA-II}$, $REMAP_3$: $REMAP_{C4.5+SPEA2}$, $REMAP_4$: $REMAP_{RIPPER+IBEA}$, $REMAP_5$: $REMAP_{RIPPER+NSGA-II}$, $REMAP_6$: $REMAP_{RIPPER+SPEA2}$, $REMAP_7$: $REMAP_{PART+IBEA}$, $REMAP_8$: $REMAP_{PART+NSGA-II}$, $REMAP_9$: $REMAP_{PART+SPEA2}$.

Table E-5 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 E-5, 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 [67].

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 E-6. 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 E-4. Moreover, Fig. E-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 E-6. Comparison of $APFD_c$ with respect to $REMAP_{RIPPER+SPEA2(2obj)}$ using the Vargha and Delaney Statistics and U test*

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
$REMAP_{4\text{-}3obj}$ vs. $REMAP_{6\text{-}2obj}$	0.23	**<0.0001**	1.00	**<0.0001**	0.97	**<0.0001**	0.24	**<0.0001**	0.92	**<0.0001**

*$REMAP_{4\text{-}3obj}$: $REMAP_{RIPPER+IBEA(3obj)}$, $REMAP_{6\text{-}2obj}$: $REMAP_{RIPPER+SPEA2(2obj)}$.

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}, G_{2obj}, and G_{3obj}. Table E-7 presents the results of comparing $REMAP_{RIPPER+SPEA2(2obj)}$ against G_{1obj}, G_{2obj}, and G_{3obj}. Based on the results from Table E-7, 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 E-4 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 E-8 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 E-4 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}.

Table E-7. Comparison of $APFD_c$ with respect to $REMAP_{RIPPER+SPEA2(2obj)}$ using the Vargha and Delaney Statistics and U test

RQ	Comparison	CS_1		CS_2		CS_3		CS_4		CS_5	
		$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value
3	G_{1obj}	1.00	**<0.0001**	1.00	**<0.0001**	1.00	**<0.0001**	0.17	**<0.0001**	0.01	**<0.0001**
	G_{2obj}	1.00	**<0.0001**	1.00	**<0.0001**	1.00	**<0.0001**	0.85	**<0.0001**	0.01	**<0.0001**
	G_{3obj}	1.00	**<0.0001**	0.99	**<0.0001**	1.00	**<0.0001**	0.85	**<0.0001**	0.01	**<0.0001**
4	$SSBP_{RIPPER+SPEA2\ (2obj)}$	1.00	**<0.0001**	0.95	**<0.0001**	1.00	**<0.0001**	0.95	**<0.0001**	0.83	**<0.0001**
	$SSBP_{RIPPER+IBEA(3obj)}$	1.00	**<0.0001**	0.09	**<0.0001**	0.98	**<0.0001**	0.96	**<0.0001**	0.50	0.862
5	$RBP_{RIPPER-2obj}$	0.82	**<0.0001**	0.89	**<0.0001**	0.60	**<0.0001**	0.86	**<0.0001**	0.85	**<0.0001**
	$RBP_{RIPPER-3obj}$	0.80	**<0.0001**	0.90	**<0.0001**	0.46	**<0.0001**	0.86	**<0.0001**	0.86	**<0.0001**

Table E-8. Comparison of $APFD_c$ with respect to $REMAP_{RIPPER+IBEA(3obj)}$ using the Vargha and Delaney Statistics and U test

RQ	Comparison	CS_1		CS_2		CS_3		CS_4		CS_5	
		$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value	$\widehat{A}_{12}$	*p*-value
3	G_{1obj}	1.00	**<0.0001**	1.00	**<0.0001**	1.00	**<0.0001**	0.01	**<0.0001**	0.61	**<0.0001**
	G_{2obj}	0.92	**<0.0001**	1.00	**<0.0001**	1.00	**<0.0001**	0.43	**<0.0001**	0.48	**0.021**
	G_{3obj}	0.92	**<0.0001**	1.00	**<0.0001**	1.00	**<0.0001**	0.43	**<0.0001**	0.49	0.184
4	$SSBP_{RIPPER+SPEA2(2obj)}$	0.94	**<0.0001**	1.00	**<0.0001**	1.00	**<0.0001**	0.86	**<0.0001**	0.98	**<0.0001**
	$SSBP_{RIPPER+IBEA(3obj)}$	1.00	**<0.0001**	0.95	**<0.0001**	1.00	**<0.0001**	0.91	**<0.0001**	0.87	**<0.0001**
5	$RBP_{RIPPER-2obj}$	0.70	**<0.0001**	1.00	**<0.0001**	0.89	**<0.0001**	0.72	**<0.0001**	0.98	**<0.0001**
	$RBP_{RIPPER-3obj}$	0.69	**<0.0001**	1.00	**<0.0001**	0.82	**<0.0001**	0.72	**<0.0001**	0.98	**<0.0001**

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 E-9 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 E-9, 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 [67].

Table E-9. Ranking of 18 variants of SSBP with two objectives and three objectives*

CS	**# of obj**	**Rank**								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
CS_1	2	$SSBP_6$	$SSBP_4$	$SSBP_5$	$SSBP_2$	$SSBP_1$	$SSBP_3$	$SSBP_7$	$SSBP_8$	$SSBP_9$
	3	$SSBP_4$	$SSBP_5$	$SSBP_6$	$SSBP_2$	$SSBP_7$	$SSBP_9$	$SSBP_8$	$SSBP_1$	$SSBP_3$
CS_2	2	$SSBP_7$	$SSBP_9$	$SSBP_8$	$SSBP_1$	$SSBP_3$	$SSBP_2$	$SSBP_4$	$SSBP_6$	$SSBP_5$
	3	$SSBP_7$	$SSBP_1$	$SSBP_4$	$SSBP_9$	$SSBP_3$	$SSBP_6$	$SSBP_8$	$SSBP_2$	$SSBP_5$
CS_3	2	$SSBP_6$	$SSBP_9$	$SSBP_3$	$SSBP_5$	$SSBP_2$	$SSBP_8$	$SSBP_1$	$SSBP_4$	$SSBP_7$
	3	$SSBP_3$	$SSBP_9$	$SSBP_4$	$SSBP_6$	$SSBP_7$	$SSBP_2$	$SSBP_1$	$SSBP_8$	$SSBP_5$
CS_4	2	$SSBP_7$	$SSBP_1$	$SSBP_3$	$SSBP_2$	$SSBP_6$	$SSBP_8$	$SSBP_5$	$SSBP_4$	$SSBP_9$
	3	$SSBP_7$	$SSBP_4$	$SSBP_3$	$SSBP_1$	$SSBP_9$	$SSBP_6$	$SSBP_5$	$SSBP_2$	$SSBP_8$
CS_5	2	$SSBP_5$	$SSBP_2$	$SSBP_6$	$SSBP_8$	$SSBP_3$	$SSBP_4$	$SSBP_1$/$SSBP_7$/$SSBP_9$		
	3	$SSBP_1$	$SSBP_3$	$SSBP_6$	$SSBP_9$	$SSBP_7$	$SSBP_4$	$SSBP_2$	$SSBP_5$	$SSBP_8$

*$SSBP_1$: $SSBP_{C4.5+IBEA}$, $SSBP_2$: $SSBP_{C4.5+NSGA-II}$, $SSBP_3$: $SSBP_{C4.5+SPEA2}$, $SSBP_4$: $SSBP_{RIPPER+IBEA}$, $SSBP_5$: $SSBP_{RIPPER+NSGA-II}$, $SBPS_6$: $SSBP_{RIPPER+SPEA2}$, $SBPS_7$: $SSBP_{PART+IBEA}$, $SSBP_8$: $SSBP_{PART+NSGA-II}$, $SSBP_9$: $SSBP_{PART+SPEA2}$.

Based on Table E-7, 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 E-8 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. E-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 RQ5. 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., $RBP_{RIPPER\text{-}2obj}$) and RS_{3obj} (i.e., $RBP_{RIPPER\text{-}3obj}$) 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 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.

Table E-10: Ranking of six variants of RBP with two objectives and three objectives

CS	Initial Solution	Rank		
		1	2	3
CS_1	RS_{2obj}	$RBP_{RIPPER\text{-}2obj}$	$RBP_{PART\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$
	RS_{3obj}	$RBP_{RIPPER\text{-}3obj}$	$RBP_{PART\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$
CS_2	RS_{2obj}	$RBP_{RIPPER\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$	$RBP_{PART\text{-}2obj}$
	RS_{3obj}	$RBP_{RIPPER\text{-}3obj}$	$RBP_{C4.5\text{-}3obj}$	$RBP_{PART\text{-}3obj}$
CS_3	RS_{2obj}	$RBP_{RIPPER\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$	$RBP_{PART\text{-}2obj}$
	RS_{3obj}	$RBP_{RIPPER\text{-}3obj}$	$RBP_{C4.5\text{-}2obj}$	$RBP_{PART\text{-}2obj}$
CS_4	RS_{2obj}	$RBP_{RIPPER\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$	$RBP_{PART\text{-}2obj}$
	RS_{3obj}	$RBP_{RIPPER\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$	$RBP_{PART\text{-}2obj}$
CS_5	RS_{2obj}	$RBP_{PART\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$	$RBP_{RIPPER\text{-}2obj}$
	RS_{3obj}	$RBP_{PART\text{-}2obj}$	$RBP_{C4.5\text{-}2obj}$	$RBP_{RIPPER\text{-}2obj}$

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 E-10 such that a lower rank implies better performance. We can observe from Table E-10 that $RBP_{RIPPER\text{-}2obj}$ and $RBP_{RIPPER\text{-}3obj}$ 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 [67].

Table E-7 and Table E-8 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., $RBP_{RIPPER\text{-}2obj}$ and $RBP_{RIPPER\text{-}3obj}$). It can be observed from Table E-7 and Table E-8 that as compared to $RBP_{RIPPER\text{-}2obj}$ and $RBP_{RIPPER\text{-}3obj}$, 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 E-4, $REMAP_{RIPPER+SPEA2(2obj)}$ 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. E-6 shows that the median $APFD_c$ score produced by $REMAP_{RIPPER+SPEA2(2obj)}$, $REMAP_{RIPPER+IBEA(3obj)}$ are higher than $RBP_{RIPPER\text{-}2obj}$, and $RBP_{RIPPER\text{-}3obj}$.

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.

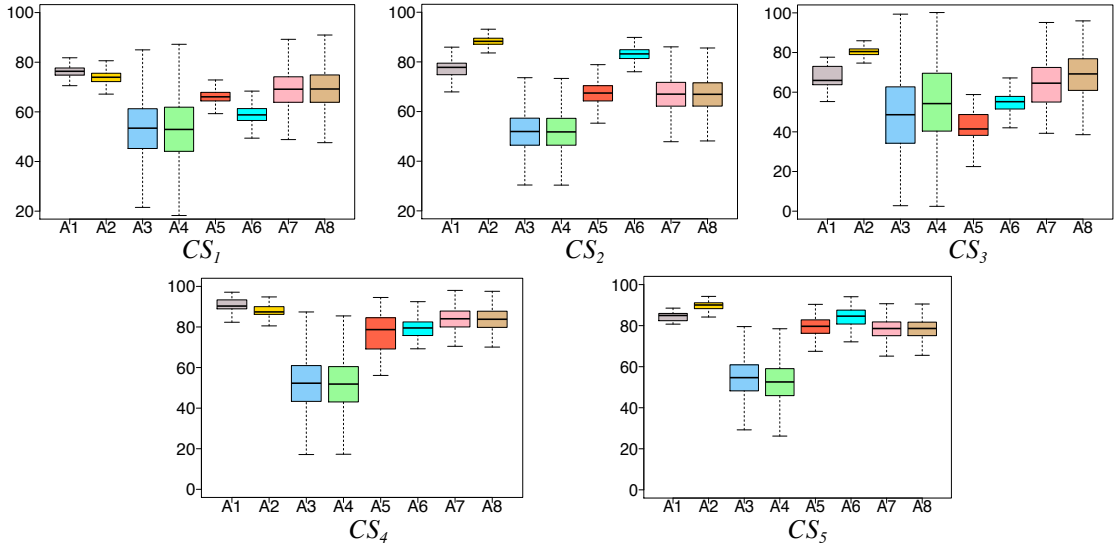

Fig. E-6. Boxplot of $APFD_c$ produced by different approaches for the case studies*

*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: $RBP_{RIPPER-3obj}$.

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 termination conditions are met. However, the variants of Greedy may get stuck in local search space and result in sub-optimal solutions [68]. 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 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. E-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. E-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* produced by G_{1obj} is worse than RS_{2obj} and RS_{3obj} (Table E-4) 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* 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 E-4) 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. E-7. However, the best variants of REMAP and 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.

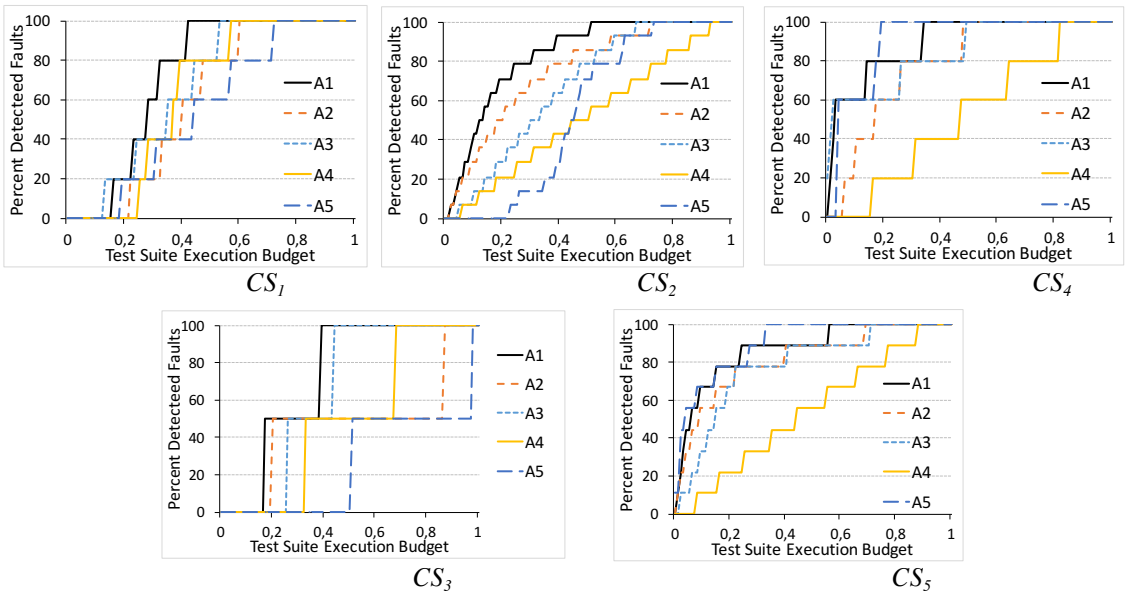

Fig. E-7. Percentage of total faults detected on executing test suite total execution budget*

A1: avg($REMAP_{RIPPER+SPEA2(2obj)}$, $REMAP_{RIPPER+IBEA(3obj)}$), A2: avg($SSBP_{RIPPER+SPEA2(2obj)}$, $SSBP_{RIPPER+IBEA(3obj)}$), A3: avg($RBP_{RIPPER-2obj}$, $RBP_{RIPPER-3obj}$), A4: avg(RS_{2obj}, RS_{3obj}), A5: avg(G_{1obj}, G_{2obj}, G_{3obj}).

8.2 Threats to Validity

The threats to *internal validity* consider the internal parameters (e.g., algorithm parameters) that might influence the obtained results [69]. In our experiments, *internal validity* threats might arise due to experiments with only one set of configuration settings for the algorithm parameters [70]. However, note that these settings are from the literature [71]. 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 [64, 72-74].

Threats to *construct validity* arise when the measurement metrics do not sufficiently cover the concepts they are supposed to measure [8]. 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 [54-56]. 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 [9, 24, 31]. 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 [24, 31].

The threats to *conclusion validity* relate to the factors that influence the conclusion drawn from the experiments [75]. 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 [57], 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 [69]. 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 [50]. 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 [41].

9 Related Work

There exists a large body of research on TP [7, 8, 26, 28, 76-78], and a broad view of the state-of-the-art on TP is presented in [79, 80]. Different kinds of literature have presented different prioritization techniques such as search-based [8, 28, 76] and linear programming based [27, 81, 82]. 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 [12]. 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 [9, 12-14, 48, 83, 84]. For instance, Kim et al. [12] 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. [9] 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. [83] 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. [16, 29] 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. [85] 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 [85] 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 [85] 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 [85]; 2) sensitive constant needs to be set up manually to prioritize/deprioritize test cases in [85], 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 [85] whereas only FDC is considered for static prioritization in [85].

9.2 Rule Mining for Regression Testing

There are only a few works that focus on applying rule mining techniques for TP [86, 87]. The authors in [86, 87] 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 [86, 87], 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 [31] 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 [31]; 2) REMAP uses a search-based test case prioritization component to obtain the static order of test case before execution, which is different than [31] 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 [31].

In our conference paper [17], 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.

Acknowledgement

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

[1] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM,* vol. 41, no. 5, pp. 81-86. 1998.

[2] E. Borjesson and R. Feldt, "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)*, pp. 350-359. IEEE, 2012.

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 6, no. 2, pp. 173-210. 1997.

[4] C. Kaner, "Improving the maintainability of automated test suites," *Software QA,* vol. 4, no. 4, 1997.

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

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

[7] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 179-188. IEEE, 1999.

[8] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering,* vol. 33, no. 4, pp. 225-237. IEEE, 2007.

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

[10] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering,* vol. 28, no. 2, pp. 159-182. 2002.

[11] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering* pp. 264-274. IEEE, 1997.

[12] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pp. 119-129. IEEE, 2002.

[13] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh, "An improved method for test case prioritization by incorporating historical test case data," *Science of Computer Programming,* vol. 78, no. 1, pp. 93-116. 2012.

[14] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 58-68. IEEE, 2015.

[15] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pp. 1493-1500. ACM, 2013.

[16] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen, "Multi-objective test prioritization in software product line testing: an industrial case study," in *Proceedings of the 18th International Software Product Line Conference*, pp. 32-41. ACM, 2014.

[17] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "REMAP: Using Rule Mining and Multi-Objective Search for Dynamic Test Case Prioritization," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2018.

[18] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 115-123. 1995.

[19] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* vol. 6, no. 2, pp. 182-197. IEEE, 2002.

[20] J. R. Quinlan, "C4. 5: Programming for machine learning," *Morgan Kauffmann,* vol. 38, p. 48. 1993.

[21] E. Frank and I. H. Witten, "Generating accurate rule sets without global optimization," 1998.

[22] E. Zitzler, M. Laumanns, and L. Thiele, "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), 2001.

[23] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *International Conference on Parallel Problem Solving from Nature*, pp. 832-842. Springer, 2004.

[24] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 12-22. ACM, 2017.

[25] S. Elbaum, A. Mclaughlin, and J. Penix. (2014). *The Google Dataset of Testing Results*. Available: https://code.google.com/p/google-shared-dataset-of-test-suite-results

[26] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 329-338. IEEE Computer Society, 2001.

[27] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering,* vol. 42, no. 5, pp. 490-505. 2016.

[28] Z. Li, Y. Bian, R. Zhao, and J. Cheng, "A fine-grained parallel multi-objective test case prioritization on GPU," in *Proceedings of the International Symposium on Search Based Software Engineering*, pp. 111-125. Springer, 2013.

[29] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "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*, pp. 182-191. ACM, 2016.

[30] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed, "Multi-objective test case prioritization in highly configurable systems: A case study," *Journal of Systems and Software,* vol. 122, pp. 287-310. 2016.

[31] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 142-151. ACM, 2014.

[32] D. J. Hand, "Principles of data mining," *Drug safety,* vol. 30, no. 7, pp. 621-622. 2007.

[33] D. T. Larose, *Introduction to data mining*: Wiley Online Library, 2005.

[34] C. C. Aggarwal, *Data mining: the textbook*: Springer, 2015.

[35] O. Maimon and L. Rokach, "Introduction to knowledge discovery and data mining," in *Data Mining and Knowledge Discovery Handbook*, ed: Springer, 2009, pp. 1-15.

[36] G. Holmes, M. Hall, and E. Prank, "Generating rule sets from model trees," in *Australasian Joint Conference on Artificial Intelligence*, pp. 1-12. Springer, 1999.

[37] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary Computation,* vol. 8, no. 2, pp. 173-195. MIT Press, 2000.

[38] A. S. Sayyad and H. Ammar, "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*, pp. 21-27. IEEE, 2013.

[39] N. Srinvas and K. Deb, "Multi-objective function optimization using non-dominated sorting genetic algorithms," *Evolutionary Computation,* vol. 2, no. 3, pp. 221-248. MIT Press, 1994.

[40] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "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*, pp. 631-642. ACM, 2016.

[41] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang, "Multiobjective evolutionary algorithms: A survey of the state of the art," *Swarm and Evolutionary Computation,* vol. 1, no. 1, pp. 32-49. 2011.

[42] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham, "AbYSS: Adapting scatter search to multiobjective optimization," *IEEE Transactions on Evolutionary Computation,* vol. 12, no. 4, pp. 439-457. IEEE, 2008.

[43] P. Hajela and C.-Y. Lin, "Genetic search strategies in multicriterion optimal design," *Structural optimization,* vol. 4, no. 2, pp. 99-107. 1992.

[44] W. Lin, S. A. Alvarez, and C. Ruiz, "Efficient adaptive-support association rule mining for recommender systems," *Data mining and knowledge discovery,* vol. 6, no. 1, pp. 83-105. 2002.

[45] W. Lin, S. A. Alvarez, and C. Ruiz, "Collaborative recommendation via adaptive association rule mining," *Data Mining and Knowledge Discovery,* vol. 6, pp. 83-105. 2000.

[46] R. J. Bayardo Jr, "Brute-Force Mining of High-Confidence Classification Rules," in *KDD*, pp. 123-126. 1997.

[47] W. Shahzad, S. Asad, and M. A. Khan, "Feature subset selection using association rule mining and JRip classifier," *International Journal of Physical Sciences,* vol. 8, no. 18, pp. 885-896. 2013.

[48] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *IEEE International Conference on Software Maintenance (ICSM)*, pp. 540-543. IEEE, 2013.

[49] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "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*, pp. 172-190. Springer, 2016.

[50] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 619-630. ACM, 2016.

[51] A. Cano, A. Zafra, and S. Ventura, "An interpretable classification rule mining algorithm," *Information Sciences,* vol. 240, pp. 1-20. 2013.

[52] A. Veloso, W. Meira Jr, and M. J. Zaki, "Lazy associative classification," in *Sixth International Conference on Data Mining (ICDM)*, pp. 645-654. IEEE, 2006.

[53] *Supplementary material*. Available: https://remap-ICST.netlify.com

[54] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Hypervolume-based search for test case prioritization," in *International Symposium on Search Based Software Engineering*, pp. 157-172. Springer, 2015.

[55] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 234-245. ACM, 2015.

[56] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk, "How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects," *IEEE Transactions on Software Engineering,* 2018.

[57] A. Arcuri and L. Briand, "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)*, pp. 1-10. IEEE, 2011.

[58] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132. 2000.

[59] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics,* pp. 50-60. JSTOR, 1947.

[60] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association,* vol. 47, no. 260, pp. 583-621. 1952.

[61] O. J. Dunn, "Multiple comparisons using rank sums," *Technometrics,* vol. 6, no. 3, pp. 241-252. 1964.

[62] C. Bonferroni, "Teoria statistica delle classi e calcolo delle probabilita," *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze,* vol. 8, pp. 3-62. 1936.

[63] A. Dinno, "Nonparametric pairwise multiple comparisons in independent groups using Dunn's test," *Stata Journal,* 2015.

[64] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*: Morgan Kaufmann, 2016.

[65] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software,* vol. 42, no. 10, pp. 760-771. Elsevier, 2011.

[66] *The Abel computer cluster*. Available: http://www.uio.no/english/services/it/research/hpc/abel/

[67] *Technical Report (2018-02)*. Available: https://www.simula.no/publications/employing-rule-mining-and-multi-objective-search-dynamic-test-case-prioritization

[68] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Software Engineering Notes,* vol. 31, no. 1, pp. 35-42. ACM, 2006.

[69] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*: John Wiley & Sons, 2012.

[70] M. de Oliveira Barros and A. Neto, "Threats to Validity in Search-based Software Engineering Empirical Studies," *UNIRIO-Universidade Federal do Estado do Rio de Janeiro, techreport,* vol. 6, 2011.

[71] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Proceedings of the 3rd International Symposium on Search Based Software Engineering*, pp. 33-47. Springer, 2011.

[72] S. Vijayarani and M. Divya, "An efficient algorithm for generating classification rules," *International Journal of Computer Science and Technology,* vol. 2, no. 4, 2011.

[73] B. Ma, K. Dejaeger, J. Vanthienen, and B. Baesens, "Software defect prediction based on association rule classification," 2011.

[74] K. Song and K. Lee, "Predictability-based collective class association rule mining," *Expert Systems with Applications,* vol. 79, pp. 1-7. 2017.

[75] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, "Experimentation in software engineering: an introduction. 2000," ed: Kluwer Academic Publishers, 2000.

[76] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 1-12. ACM, 2006.

[77] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pp. 523-534. IEEE, 2016.

[78] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "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*, pp. 367-378. IEEE, 2017.

[79] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability,* vol. 22, no. 2, pp. 67-120. Wiley Online Library, 2012.

[80] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal,* vol. 21, no. 3, pp. 445-478. 2013.

[81] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pp. 213-224. ACM, 2009.

[82] S. Mirarab, S. Akhlaghi, and L. Tahvildari, "Size-constrained regression test case selection using multicriteria optimization," *IEEE Transactions on Software Engineering,* vol. 38, no. 4, pp. 936-956. 2012.

[83] H. Park, H. Ryu, and J. Baik, "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*, pp. 39-46. IEEE, 2008.

[84] Y. Fazlalizadeh, A. Khalilian, M. Abdollahi Azgomi, and S. Parsa, "Incorporating historical test case performance data and resource constraints into test case prioritization," *Tests and Proofs,* pp. 43-57. 2009.

[85] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," in *Proceedings of the 31st Annual Computer Software and Applications Conference (COMPSAC)*, pp. 465-474. IEEE, 2007.

[86] P. Mahali, A. A. Acharya, and D. P. Mohapatra, "Test Case Prioritization Using Association Rule Mining and Business Criticality Test Value," in *Computational Intelligence in Data Mining—Volume 2*, ed: Springer, 2016, pp. 335-345.

[87] A. A. Acharya, P. Mahali, and D. P. Mohapatra, "Model based test case prioritization using association rule mining," in *Computational Intelligence in Data Mining-Volume 3*, ed: Springer, 2015, pp. 429-440.