EPiC Series in Computing

Volume 76, 2021, Pages 105–116

SEDE 2020. 29th International Conference on Software Engineering and Data Engineering

# Multi-Objective Regression Test Selection

Yizhen Chen and Mei-Hwa Chen

SUNY at Albany, New York, U.S.A.
(ychen33, mchen)@albany.edu

**Abstract**

Regression testing is challenging, yet essential, for maintaining evolving complex software. Efficient regression testing that minimizes the regression testing time and maximizes the detection of the regression faults is in great demand for fast-paced software development. Many research studies have been proposed for selecting regression tests under a time constraint. This paper presents a new approach that first evaluates the fault detectability of each regression test based on the extent to which the test is impacted by the changes. Then, two optimization algorithms are proposed to optimize a multi-objective function that takes fault detectability and execution time of the test as inputs to select an optimal subset of the regression tests that can detect maximal regression faults under a given time constraint. The validity and efficacy of the approach were evaluated using two empirical studies on industrial systems. The promising results suggest that the proposed approach has great potential to ensure the quality of the fast-paced evolving systems.

## 1 Introduction

Most software systems are continuously evolving to better serve user or market needs. When a change is made to software or its execution context, regression testing should be performed to ensure no regression fault is causing previously working features to fail. Although re-executing all the test cases can reconfirm the working of the non-modified features, it can be time-consuming when new tests are continually added to the test suite, and even worse, it may not be feasible when the time to redeploy is limited. To minimize the maintenance time to prevent service disruption of production software, an efficient and effective selection from the regression test suite to perform regression testing is essential.

Techniques for improving the efficacy of regression testing have been well studied. The test suite minimization approaches [10, 15, 18, 20] aim at removing obsolete test cases that are no longer valid for the changed software and redundant test cases to form a minimum test suite that satisfies the test requirements. The prioritization approaches [11, 22, 24] intend to determine an order of test executions that can achieve the desired goal early, such that if the testing process terminates early then the test cases that have potentially higher values such as higher code coverage or estimated fault detection rate would be executed before the termination. The minimization and the prioritization methods can reduce the number of test cases to be retested, but they may overlook some fault-revealing test cases [21]. The selective regression testing approaches [25, 5, 7, 10, 26, 21] identify modification-traversing test cases that execute the modified part of the program. Under a controlled environment, these approaches are safe and can select all fault-revealing test cases. However, they may select many test cases that cannot be executed within the time limit, and it can be unsafe if the execution context is changed.

When executing all or selective regression tests cannot be done within the given time constraint, it is strongly desired that if the changed software is to fail then it should fail in very few test executions, so if there are any regression faults they can be corrected immediately. Taking the time constraint into account, several multi-objective regression testing approaches have been proposed [27, 13, 6, 23], which apply multiple criteria to select a closed to an optimum subset of test cases that account for the selected criteria to obtain a maximal efficacy. These approaches apply various optimization algorithms to find the best solution to balance the trade-offs between the cost and code coverage. The existing approaches estimate the fault detectability of the regression test case by code coverage, such as block or function coverage, or fault detectability based on history. However, regression faults are introduced by the changes, which do not exist before the changes and are relevant to the modified and affected parts of the software only. It has been suggested that a test case is fault-revealing only if it is modification-traversing [21]. Therefore, a test case that has a high code coverage but does not execute any modified or affected code, i.e., not modification-revealing, will not be fault-revealing. Therefore, it is not clear that the techniques based on code coverage or history can be effective for detecting regression faults.

This paper presents a new approach that formulates a multi-objective function striving to obtain an optimal solution to maximize the fault detection rate under a time constraint. Our approach focuses on selecting the tests impacted most by the changes in the program, upgrade/downgrade of the library, the changes in databases or configuration files. We use program states before and after an invocation of a function to determine if the function is affected by the change and can potentially have regression faults. A program state before a function call denoted in the precondition of a function, including the properties required for the successful execution of the function and the postcondition of a function, indicates the state of the program after the execution of the function.

To maximize the fault detectability within a constrained time, we model the test selection as a multi-objective Knapsack problem, which determines the number of each test case to select so that the total execution time does not exceed the time limit while maximizing the number of fault-revealing tests (failed tests). We apply two optimization algorithms commonly used to solve the sensor placement problems to obtain an optimal set of test cases. To evaluate the effectiveness of the proposed approach, we conducted three case studies on three industrial systems and real regression faults. The results suggest that our approach is much more effective than the existing code coverage-based approaches.

The contributions of this paper are:

1. Our multi-objective test selection takes the coverage of affected functions, modified functions, and test execution time as the objective, which will select the test cases that are more likely to expose regression faults.

2. Our approach can be fully automated, and we have implemented a prototype to conduct the empirical studies. The results show that our approach significantly reduces the size of the regression tests while selecting all the fault-revealing regression tests. Thus, regression testing can be performed efficiently and effectively to maintain the quality of the software after changes.

2. The subject programs and the faults used in the studies are real industrial programs and real regression faults. These regression faults were identified along with the evolution processes, which demonstrates the feasibility of applying the proposed approach to the real-life industrial

systems.

The remainder of this paper is organized as follows: the proposed approach is described in Section 2. Section 3 presents the two case studies and Section 4 gives an overview of the related techniques. The conclusions and future work are given in Section 5.

## 2 METHODOLOGY

The goal of our test selection technique is to identify test cases executed successfully before the change of the program or the execution context, but they potentially can fail because of the changes. To evaluate change impact on the regression tests, modification-revealing tests have been suggested for their potential to detect regression faults [21]. Thus, we strive to select the modification-revealing tests that are most likely to detect regression faults. Our approach differs from the existing techniques in (1) it considers the change impact from both program changes and execution context changes, and (2) it focuses on the change impact coverage, not on the code coverage.

The basic unit of the investigation in our approach is function; we used the affected functions to investigate if a test case is likely to be fault-revealing. To determine if a function is affected by the change, we annotate pre- and postconditions of each function to detect change impact. The precondition describes the state of the program before the execution of a function, which includes the conditions (type and value range) of the input parameters, global variables and class attributes used in the function, and the callees including functions, library, and external APIs. The postcondition of a function depicts the state of the program after the execution of the function, which includes the conditions (type and value range) of the outputs, global variables, and class attributes manipulated by the function.

The pre- and postconditions can be annotated by the assertions obtained from static analysis or created by the engineers. To automatically obtain pre- and postconditions, we developed an annotator that parses the given program to instrument the pre- and postconditions of each function. The current version of our annotator includes a Java-instrumentor adopted from Daikon [12], and a JavaScript-instrumentor adopted from Fondue [2], which parses the source code and instrument program states to the pre- and postconditions of each function. An program state is denoted as an ordered pair $< p, v >$, where $p$ is the program point, $v$ is a set of variables or function names.

Given a program $P$ and a regression test suite $T$, we first run the annotated program on $T$ to create an program states traceability matrix (ITM), where a row in ITM is associated with an program states that shows the occurrence of the program states in each test in $T$, and a column is associated with a test case that shows the program states included in the test case. The program states are indexed according to the precondition and postcondition of each function call, including the program functions, library functions, and external APIs. Additionally, we add the caller's name in the precondition to record the caller-callee relationship. ITM can be created during the testing phase when the test cases are created and executed. It is incrementally updated when there are changes in the program states after a new test is executed.

### 2.1 Change Impact Analysis

Given a program $P$, a regression test suite $T$, the modified program $P'$, the execution context $EC$, and $ITM$.

(1) If a function $f$ in $P$ is changed to $f'$ in $P'$, then every test $t$ in $T$ that executes $f$ is modification-traversing. An unmodified function g in $P$ and $P'$ is affected if there is at least

one program state $i$ in the precondition of $g$ and $i$ is in the postcondition of $f$. This implies that at least one variable used in $g$ is manipulated in $f$; thus $i$ and $g$ are affected.

(2) When a new function $g$ is added to $P'$, some functions in $P$ will be modified in $P'$ to call $g$.

(3) If a function $f$ in $P$ is deleted, then every function $g$ in $P$ that calls $f$ needs to be modified.

(4) If an attribute $a$ (a variable) in $P$ is changed (type or name change) in $P'$, then every function $f$ that uses $a$ ($a$ is in the precondition of $f$) and the program states associated with $a$ in the precondition of $f$ will be affected.

(5) If a changed function $l$ is in a library $L$ or is an external API in $EC$, then every function $f$ in $P$ that calls $l$ ($l$ is in the precondition of $f$) and the program states associated with $l$ will be affected.

(6) Most modern software applications use Object Relational Mapping to create relationships between database entities and program entities. If a database schema is changed, then all the entity objects associated with the changed table are affected. If a database entity is changed (type, name, or constraint change), then every function $f$ that references the entity object $o$ ($o$ is in the precondition of $f$) associated with the changed database entity and the program states associated with $o$ will be affected. If ORM is not used by the program, then an additional effort will be required by using the parser to capture the variables used in the SQL statements.

(7) If a setting in a configuration file is changed (name, or value if the variable denotes a path or URL), any variable $v$ in the program that refers to this setting is affected. Then every function that uses this affected variable $v$ ($v$ is in the precondition of $f$) and the program states associated with $v$ will be affected.

## 2.2 Multi-Objective Regression Test Selection

Our goal in the multi-objective regression test selection is to maximize the number of regression faults that can be detected with the selected test cases and to ensure that the cost of the test execution time will not exceed the budgeted time constraint. To model this problem, we adopt the model-based sensor placement approach to develop efficient and effective algorithms for selecting the test cases. The criteria we use include the coverage of the affected program states, modified functions, and execution time. The aim is to select the test cases that have a high number of affected program states, cover more modified functions, and have low execution time.

Given the input domain $\mathbb{T}$ and the objectives defined as above, the goal is to find a set of test cases $S$ from $\mathbb{T}$ that have minimal overall detection risk $F(S)$ that accounts for the coverage, and minimal overall cost $C(S)$, which is the cost of the test execution time, subject to a budget constraint on $C(S)$: $C(S) \leq K$, where $K$ is the time allocated for the regression testing. This is a multi-criterion optimization problem and the scalarization approach [8] is commonly used to find such Pareto-optimal solutions. In particular, we optimize the following problem

$$\min_{S \in \mathbb{T}} \quad O(S) = F(S) + \lambda C(S) \quad s.t. \quad C(S) \leq K \tag{1}$$

by choosing the appropriate weight $\lambda > 0$. All possible Pareto-optimal solutions to the minimization of $O(S)$ subject to the budget constraint $C(S) \leq K$ can be obtained by varying the weight $\lambda$. Problem (1) is a hard-combinatory optimization problem, and an exhaustive search of the optimal set is usually unfeasible. We note that there is no unique form of the objective function $O(S)$, which varies depending on its application domain. Hence, we design customized algorithms by exploring the specific structure of the problem under different situations.

We applied two algorithms to solve the problem: Algorithm 1: a sub-modular for test selection that is a combinatorial search based on sub-modular optimization. Algorithm 2: a projected gradient descent algorithm for test selection which is a graph-structured convex optimization based on continuous relaxation. Problem (1) is called a sub-modular minimization problem subject to knapsack constraints if there is no network topology constraint and can be solved in nearly linear time with the approximation factor $(1-1/e) \approx 0.632$ using the greedy algorithm [19], as shown in Algorithm 1. This algorithm starts from an empty set $S = \emptyset$, and adds the element maximizing the discrete derivative $\Delta(s|S) = O(S \cup s) - O(S)$ (line 3 and line 4). The approximation factor is defined as $O(S)/O(S^*)$ where $S$ refers to the approximated set of program states, and $S^*$ refers to the optimal set.

Furthermore, we want to consider the function interaction relationship to identify which affected function or modified function has the highest coupling with the other functions. For example, a function has the highest number of callers. If the function is affected, then the more calling contexts the function has, the more likely the regression faults, if they exist, in this function can be detected. Under this assumption, we utilize the function call graph in each test as a network graph, where a node represents a function and an edge denotes a call relationship and select the nodes that have higher incoming edges. By considering the network graph constraint, the problem becomes neither super-modular nor sub-modular. We can instead apply convex optimization techniques to solve problems (1) based on continuous relaxation. We first define a vector format of $S$ as $z \in \{0,1\}^{|\mathbb{U}|}$, and $S = \text{supp}(z)$ as the set of nonzero entries in $z$. The two objective functions $F(S)$ and $C(S)$ can be reformulated as the functions based on $z$ as $f(z)$ and $c(z)$, respectively. The constraint $S \in \mathbb{T}$ can be reformulated as $\text{supp}(z) \in \mathbb{T}$. Problem (1) can then be reformulated as:

$$\min_{z \in \{0,1\}^{|\mathbb{U}|}, \text{supp}(z) \in \mathbb{T}} o(z) = f(z) + \lambda c(z), \ \ s.t. \ c(z) \leq K, \tag{2}$$

which can be approximated using graph-structured convex optimization techniques [9, 16], where "graph-structured" means that the constraint $\text{supp}(z) \in \mathbb{T}$ is defined based on graph topology. Algorithm 2 shows the basic steps of the projected gradient descent approach for solving problem (2), where $\nabla f(z)$ refers to the gradient of $f(z)$ with respect to $z$, and $\nabla c(z)$ is defined similarly.

The vector $\nabla f(z^{(i)}) + \lambda \cdot \nabla c(z^{(i)})$ calculated in line 4 is the gradient of the overall objective function $o(z)$. The vector $b^{(i)}$ is the gradient descent update, which is the same as the update of the standard gradient descent algorithm, and $\lambda$ refers to the step size. As $b^{(i)}$ is often not within the graph-structured domain $\mathbb{T}$, line 5 is to project $b^{(i)}$ to the input domain $\mathbb{T}$ based on the following projection operator:

$$T(b^{(i)}) = \min_{z \in \{0,1\}^{|\mathbb{U}|}, \text{supp}(z) \in \mathbb{T}} \|z - b^{(i)}\|_2^2 \ \ s.t. \ \ c(z) \leq K, \tag{3}$$

where $\text{supp}(z)$ refers to the set of non-zero entries in $z$.

**Algorithm 1** A submodular algorithm for test selection

```
Require: the ground set of test cases $\mathbb{T}$.
Ensure: S
1: $S = \emptyset$
2: while $C(S) < K$ do
3:     $s = \underset{s \in \mathbb{T}}{\text{argmax}}\ O(S) - O(S \cup s) \quad s.t.\ S \cup s \in \mathbb{T}$
4:     $S = S \cup \{s\}$
5: end while
```

**Algorithm 2** A projected gradient descent algorithm for test selection

```
Require: Network $\mathbb{G}$ and the ground set of program states $\mathbb{U}$.
Ensure: S
1: $i = 0$
2: $z^{(i)} = 0$
3: repeat
4:     $b^{(i)} = z^{(i)} - \big(\nabla f(z^{(i)}) + \lambda \cdot \nabla c(z^{(i)})\big)$
5:     $z^{(i+1)} = \mathrm{T}(b^{(i)})$
6:     $i = i + 1$
7: until Convergence
8: $S$=supp($z^i$)
```

## 3 CASE STUDIES

We conducted two case studies to evaluate the feasibility and effectiveness of the proposed approach. The research questions we asked in these studies are: (1) How efficiently can Algorithm 1 detect all the regression faults as compared to the existing approaches? (2) How efficiently can Algorithm 2 detect all the regression faults with the help of the network graph? These studies were conducted on three industrial systems with real test cases and regression faults. We applied our two algorithms on these programs to compare the effectiveness of selecting fault-revealing tests and fault detectability. In addition, we computed the average of fault age (the number of regression tests executed to detect the regression fault) suggested by Kim and Porter [17] for measuring the effectiveness of regression test selection, by using different objectives.

At the beginning of these studies, we annotated the program to instrument the pre- and postconditions for each function and ran the annotated program with the test suite to create ITM. We used the coverage of modified functions, modification revealing program states, function calls, and execution time as the criteria to form the objective function. We first define:

$$Risk\_Modified\_Fn(s) = \frac{total\_number\_of\_uncovered\_modified\_function}{total\_number\_of\_modified\_function}$$

$$Risk\_Affected\_Program\_States(s) = \frac{total\_number\_of\_uncovered\_affected\_program\_states}{total\_number\_of\_affected\_program\_states}$$

$$Function\_Call\_Coverage(S) = \frac{total\_number\_of\_uncovered\_function\_calls}{total\_number\_of\_function\_calls}$$

We developed three Multi-Objective functions:
**Algorithm 1a** uses Algorithm 1 to get a set of test cases that minimizes the objective function.

$$F(s) = Risk\_Modified\_Fn(s) + Risk\_Affected\_Program\_States(s)$$

$$C(s) : The\ sum\ of\ the\ execution\ time\ for\ each\ test\ case$$

$$O(S) = F(s) + \lambda C(s)$$

Table 1: The results of the case study I.

| Case | | Algorithm 1a | | | | | | Algorithm 1b | | | | | | Algorithm 2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5% | 10% | 20% | 30% | 40% | 50% | 5% | 10% | 20% | 30% | 40% | 50% | 1 | 2 |
| 1 | $f\_t$ | 5 | 5 | 5 | 5 | 9 | 9 | 1 | 1 | 2 | 4 | 4 | 4 | 5(29.8%) | 9(34.5%) |
| | $r\_f$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | $f\_t$ | 8 | 16 | 22 | 37 | 86 | 97 | 3 | 17 | 47 | 51 | 77 | 79 | 89(41.2%) | 98(56.1%) |
| | $r\_t$ | 2 | 3 | 3 | 3 | 3 | 3 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | $f\_t$ | 19 | 39 | 68 | 89 | 140 | 147 | 11 | 29 | 61 | 93 | 115 | 121 | 140(43.9%) | 150(58.5%) |
| | $r\_f$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | $f\_t$ | 5 | 11 | 43 | 58 | 60 | 61 | 0 | 5 | 9 | 25 | 42 | 46 | 60(40.3%) | 66(52.4%) |
| | $r\_f$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | $f\_t$ | 1 | 3 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 3(8.5%) | N/A |
| | $r\_f$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | N/A |

**Algorithm 1b** uses code coverage and execution time as the objectives and applies Algorithm1.
**Algorithm 2** uses Algorithm 2 to find an optimal set of test cases that minimizes the objective function.

$F(s) = Risk\_Modified\_Fn(s) + Risk\_Modification\_Revealing\_Program\_States(s) + Function\_Call\_Coverage(s)$

$C(s) = the\ sum\ of\ the\ execution\ time\ for\ each\ test\ cases$

$O(s) = F(s) + \lambda C(s)$

The cost function was modeled at 5%, 10%, 20%, 30%, 40%, and 50% of the total execution time of the regression test suite. Algorithm 2 used the projected gradient descent algorithm, which obtained a local optimal point when it converged. After its first convergence, we re-calibrated the objective function to obtain the next optimal point.

## 3.1 Case Study I

The first study was conducted on an education software, Idea Thread Mapper [3], used in k-12 schools located in four countries, including the US, Canada, Singapore, and Taiwan. The system is built based on the microservices and is implemented in JavaScript, D3.js, Java, and uses MySQL. The current version contains 39 Java classes, 1,085 functions, and 142,908 lines of code. At the beginning of the study, there were 397 test cases created by the testing group, and in the end there were 412 test cases after removing obsolete test cases and adding new test cases during this study. Each test case takes several minutes to an hour to execute. There is often very limited time for regression testing and retesting all the test cases will delay the restoring of the services and may disrupt the classroom use, which is not viable. There were five corrective activities during this study, including the fixes after a missing database table, a change to a global variable, a change to a session, a change of URL in the configuration, and a change of library. The results of this study are summarized in Table 1, where $f\_t$ denotes the total number of selected fault-revealing tests and $r\_f$ denotes the total number of detected regression faults.

In Case 1, there were nine fault-revealing tests and one regression fault. Algorithm 2 used 29.8% of the cost and detected the regression fault, and used 34.5% of the cost selecting all the fault-revealing tests; Algorithm 1a used 5% of the cost detecting the regression fault and selected all the fault-revealing tests with 40% of the cost. Algorithm 1b also detected the regression fault with 5% of the cost, but it only selected four fault-revealing tests with 50% of the cost. The average fault age for Algorithm 1a and 1b was 12 and 18, respectively.

In Case 2, there were 98 fault-revealing tests and three regression faults. Algorithm 1a and Algorithm 1b detected all three regression faults by using 10% of the cost. Algorithm 2 used 41.2% of the cost to detect the three faults. For the fault-revealing tests, Algorithm 1a selected 97 tests and Algorithm 1b selected 79 tests with 50% of the cost. Algorithm 2 selected all 98 tests with 56.1% of the cost. The average fault age for Algorithm 1a and 1b was 6 and 8, respectively.

In Case 3, there were 150 fault-revealing tests and one regression fault. Algorithm 1a and Algorithm 1b detected the regression fault by using 5% of the cost. Algorithm 2 used 43.9% to detect the fault. For the fault-revealing tests, Algorithm 1a selected 147 tests and Algorithm 1b selected 121 tests with 50% of the cost. Algorithm 2 selected all 150 tests at 58.5% of the cost. The average fault age for Algorithm 1a and 1b was 2 and 3, respectively.

In Case 4, there were 66 fault-revealing tests and one regression fault, which was detected by Algorithm 1a at 5% of the cost, by Algorithm 1b at 10% of the cost, and by Algorithm 2 at 40.3% of the cost. For the fault-revealing tests, Algorithm 1a selected 61 tests and Algorithm 1b selected 46 tests at 50% of the cost, and Algorithm2 selected all 66 test cases at 52.4% of the cost. The average fault age for Algorithm 1a and 1b was 8 and 34, respectively.

In Case 5, there were three fault-revealing tests and one regression fault. Algorithm 1a detected the regression fault at 5% of the cost, and Algorithm 2 detected the fault at 8.5% of the cost. Algorithm 1b failed to detect the fault with 50% of the cost. All three fault-revealing tests were selected by Algorithm1a at 10% of the cost and by Algorithm 2 at 8.5% of the cost. Algorithm 1b did not select any of them at 50% of the cost. The average fault age for Algorithm 1a and 1b was 1 and 223, respectively.

**Discussion:** In this case study, we observed that Algorithm 1a performed much better than Algorithm 1b for detecting regression faults and selecting fault-revealing tests. Algorithm 2 was able to select all the fault-revealing tests with 8.5% to 58.5% of the cost. It detected all the regression faults but used more test cases than Algorithm 1a; when there were large numbers of the affected functions and modification revealing program state, Algorithm 2 required a long time to converge and obtain a local optimum. Algorithm 1b used function coverage and failed to detect the regression fault within the budget in Case 5. It is suggested that if it is desired to find a regression fault early and fix it immediately, then Algorithm 1a will be the better choice. When the regression testing is to be performed automatically to find all the fault-revealing tests within the budget, then Algorithm 2 will be the better choice.

## 3.2 Case Study II

The second case study was conducted on two health-related projects developed for the National Cancer Institute [4]. The first project was Microarray, a service that analyzes biology data and provides visualization of the analyzed results. It is a web application powered by Node.js, React.js, and R programming language. There are 21,409 lines of code, 40 functions, and 84 test cases.

A program issue was reported in JIRA due to a change in one of the APIs that returns data and its schema. The program receives the data and uses the schema to extract the data for visualization. The API changed data schema, which introduced one regression fault in the search function (Case 1) and one in the sorting function (Case 2). Among the 84 test cases.

Table 2: The results of the case study II - Microarray.

| Case | | Algorithm 1a | | | | | | Algorithm 1b | | | | | | Algorithm 2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5% | 10% | 20% | 30% | 40% | 50% | 5% | 10% | 20% | 30% | 40% | 50% | 1 | 2 |
| 1 | $f\_t$ | 1 | 2 | 3 | 5 | 7 | 9 | 0 | 0 | 0 | 1 | 3 | 3 | 7(38.1%) | 9(42.3%) |
| | $r\_f$ | 1 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| 2 | $f\_t$ | 1 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 2(8.2%) | N/A |
| | $r\_f$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | N/A |

The results are summarized in Table 2.

In Case 1, there were nine fault-revealing tests and two regression faults. Algorithm 2 used 42.3% of the cost to select all the fault-revealing tests. Algorithm 1a selected all the fault-revealing tests within 50% of the cost. Algorithm 1b only selected three tests with 50% of the cost. For the two regression faults. Algorithm 1a detected one regression fault by using 5% of the cost and two regression faults by using 10% of the cost. Algorithm 1b used 30% of the cost to detect one regression fault and failed to detect the second regression fault by 50% of the cost. Algorithm 2 used 38.1% of the cost to detect the first fault and 42.3% of the cost to detect both faults. The average fault age for Algorithm 1a and 1b was 2 and 22.5, respectively.

In Case 2, there were only two fault-revealing tests. Algorithm 1a selected both tests with 10% of the cost and Algorithm 2 used 8.2%. Algorithm 1b only selected one tests within the budget. Algorithm 1a used 5% of the cost to detect the regression fault, Algorithm 2 used 8.2%, and Algorithm 1b used 30% of the cost. The average fault age for Algorithm 1a and 1b was 1 and 42, respectively.

The second project was the Cancer Epidemiology Descriptive Cohort Database (CEDCD) [1], which contained descriptive information about cohort studies that follow groups of persons over time for cancer incidence, mortality, and other health outcomes. The CEDCD program facilitates collaboration and highlights the opportunities for research within existing cohort studies. It is a web application maintained by the Epidemiology and Genomics Research Program (EGRP), located in the Division of Cancer Control and Population Sciences, National Cancer Institute's (NCI's), National Institutes of Health.

The program is powered by Node.js, React.js, and uses MySql database. It has 32,796 lines of code, 155 functions, and 190 test cases. Two modifications that introduced the regression faults were reported in JIRA. The first change was a business logic change, which modified a stored procedure and changed its output. The return value was a data table that has a column to indicate genders. Before the change, the gender was an integer, -1(unknown), 0(both), 1(male), -1(female). After the change, the gender was changed into a String Type as male, female, unknown, and both, which caused one test case to fail. A second change was made to a drop-down selection box, which changed "no cancer" to "No Cancer". This change caused five test cases to execute a function that takes "no cancer" as input to fail. The results are summarized in Table 3.

In Case 1, there were five fault-revealing tests and one regression fault. Algorithm 2 used 26.5% of the cost to select all the fault-revealing tests. Algorithm 1a used 30% and Algorithm 1b only selected three tests within the 50% budget. The regression fault was detected by Algorithm 1a with 5% of the cost, Algorithm 1b with 30%, and Algorithm 2 with 21.3%. The average fault age for Algorithm 1a and 1b was 8 and 65, respectively.

In Case 2, there were only one fault-revealing test and one regression fault. Algorithm 1a used 5% of the cost to select the test and detect the regression fault. Algorithm 1b used 30% and Algorithm 2 used 2% of the cost to select the test and detect the regression fault. The

Table 3: The results of the case study II - CEDCD.

| Case | | Algorithm 1a | | | | | | Algorithm 1b | | | | | | Algorithm2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5% | 10% | 20% | 30% | 40% | 50% | 5% | 10% | 20% | 30% | 40% | 50% | 1 | 2 |
| 1 | $f\_t$ | 1 | 1 | 3 | 5 | 5 | 5 | 0 | 0 | 0 | 1 | 1 | 3 | 4(21.3%) | 5(26.5%) |
| | $r\_f$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | $f\_t$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1(2.6%) | N/A |
| | $r\_f$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | N/A |

average fault age for Algorithm 1a and 1b was 1 and 76, respectively.

**Discussion:** In this case study, the number of the affected functions and program states was much less than the ones in the Case study I, and we observed that Algorithm 1a performed much better than Algorithm 1b for selecting fault-revealing tests and detecting regression faults. Algorithm 2 converged faster in this study than in Case study 1. For the regression fault detection, the performance of Algorithm 1a and Algorithm 2 was about the same. Thus, Algorithm 2 will be the better choice when the impact of the change is small.

## 4 Related Work

A number of multi-objective regression testing techniques have been proposed. Yoo and Harman [27, 14] introduced a multi-objective regression test optimization (MORTO) framework and proposed a Pareto efficient approach that includes a two-objective formulation combining code coverage and cost, and a three-objective formulation combining code coverage, cost, and fault history. They evaluated the effectiveness of the three algorithms used to solve these two multi-objective problems. In addition, they provided an approach(algorithms) to optimize multi-objective formulation with conflicting constraints. The constraints (code coverage) can be used to limit the type of program's modification that can be covered.

Garousi et al. [13] adopted MORTO framework and developed a customized genetic algorithm to provide full coverage of the affected (changed) requirements while considering multiple cost and benefit factors such as minimizing the number of test cases and maximizing the cumulative number of detected faults by each test suite. However, the affected requirements are only associated with code changes and no context change is addressed.

Anwar and Ahsan [6] applied Fuzzy Logic to optimize regression test selection; they used fault detection rate, execution time, requirement coverage, and the impact of failure requirements as the objectives and applied to two case studies.

Sampath et al. [23] presented a uniform representation of hybrid criteria. They formulated three hybrid criteria, including rank, merge, and choice, and observed that these hybrid criteria outperformed their constituent individual criterion. Their studies show that a hybrid criterion that combines several individual criteria performs better than a single criterion. But the effectiveness depends on which criteria are used.

In summary, the existing multi-objective regression test selection approaches apply various factors to select the test cases that are most likely to be modification-revealing with minimum cost, but they only consider the modifications made to the code, not to the execution context.

## 5 Conclusions and Future Work

We have presented two algorithms to select a close to an optimum set of test cases that have the highest potential fault detection capability while keeping the execution time within the budget constraint. Our approach considers impact from changes in the program as well as in its execution context, and our multi-objective function selects an optimal subset of test cases that are likely fault-revealing and the total execution time is within the given time constraint. The results of our case studies show that by focusing on the affected functions, our approach is much more effective than the existing approaches to detect regression faults under the time constraint.

To improve the applicability of the proposed approach, we are working on defining program states for other languages. To increase precision, we are studying what types of program state are more sensitive to the changed behavior and aim to only use these program states to improve precision.

## References

[1] Cedcd. `https://cedcd.nci.nih.gov/home`.

[2] fondue. `https://github.com/adobe-research/fondue`.

[3] Idea thread mapper. `http://www.idea-thread.org`.

[4] Nci. `https://www.cancer.gov`.

[5] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*, pages 348–357. IEEE, 1993.

[6] Z. Anwar and A. Ahsan. Multi-objective regression test suite optimization with fuzzy logic. In *Multi Topic Conference (INMIC), 2013 16th International*, pages 95–100. IEEE, 2013.

[7] T. Ball. On the limit of control flow analysis for regression test selection. *ACM SIGSOFT Software Engineering Notes*, 23(2):134–142, 1998.

[8] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[9] F. Chen and B. Zhou. A generalized matching pursuit approach for graph-structured sparsity. In *IJCAI*. Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, 2016.

[10] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[13] V. Garousi, R. Özkan, and A. Betin-Can. Multi-objective regression test selection in practice: An empirical study in the defense software industry. *Information and Software Technology*, 2018.

[14] M. Harman. Making the case for morto: Multi objective regression test optimization. In *2011 Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 111–114. IEEE, 2011.

[15] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[16] C. Hegde, P. Indyk, and L. Schmidt. A nearly-linear time framework for graph-structured sparsity. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 928–937, 2015.

[17] 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*, pages 119–129, 2002.

[18] J.-W. Lin, C.-Y. Huang, and C.-T. Lin. Test suite reduction analysis with enhanced tie-breaking techniques. In *Management of Innovation and Technology, 2008. ICMIT 2008. 4th IEEE International Conference on*, pages 1228–1233. IEEE, 2008.

[19] Y. H. Lucas Bordeaux and P. Kohli. *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2014.

[20] N. Mansour and K. El-Fakih. Simulated annealing and genetic algorithms for optimal regression testing. *Journal of Software: Evolution and Process*, 11(1):19–34, 1999.

[21] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[22] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[23] S. Sampath, R. Bryce, and A. M. Memon. A uniform representation of hybrid criteria for regression testing. *IEEE transactions on software engineering*, 39(10):1326–1344, 2013.

[24] 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*, pages 1–12. ACM, 2006.

[25] L. White and K. Abdullah. A firewall approach for regression testing of object-oriented software. *Software Quality Week*, 27, 1997.

[26] L. J. White and H. K. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 262–271. IEEE, 1992.

[27] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.