2017 24th Asia-Pacific Software Engineering Conference

A Regression Test Case Prioritization Algorithm Based on Program Changes and Method Invocation Relationship

Wenhao Fu[1,2], Huiqun Yu[1], Guisheng Fan[1], Xiang Ji[1], Xin Pei[3]
1 Department of Computer Science and Engineering
2 Shanghai Key Laboratory of Computer Software Evaluating and Testing
East China University of Science and Technology
3 The Third Research Institute of the Ministry of Public Security
Shanghai 200237, China
Corresponding Author: yhq@ecust.edu.cn

***Abstract*—Regression testing is essential for assuring the quality of a software product. Because rerunning all test cases in regression testing may be impractical under limited resources, test case prioritization is a feasible solution to optimize regression testing by reordering test cases for the current testing version. In this paper, we propose a new test case prioritization algorithm based on program changes and method (function) invocation relationship. Combining the estimated risk value of each program method (function) and the method (function) coverage information, the fault detection capability of each test case can be calculated. The algorithm reduces the prioritization problem to an integer linear programming (ILP) problem, and finally prioritizes test cases according to their fault detection capabilities. Experiments are conducted on 11 programs to validate the effectiveness of our proposed algorithm. Experimental results show that our approach is more effective than some well studied test case prioritization techniques in terms of average percentage of fault detected (APFD) values.**

Keywords—regression testing; test case prioritization; program changes; method invocation

I. INTRODUCTION

Regression testing is an essential, but expensive process in the software development that guarantees software quality. It must be performed to validate program changes when a program is modified each time. However, due to the limitation of resources and time, reusing all test cases is impractical. Hence, many techniques for improving regression testing have been investigated. These techniques may include, for instance, selecting only a subset of the original test suite based on modified information to achieve test target[1, 2], reducing the size of test suite by identifying and eliminating redundant test cases [3-5] , or reordering test cases such that they can meet testing goals earlier[3, 6-9]. However, the former two techniques may be not safe because they have omitted test cases that could detect faults, while another one will not.

Test case prioritization (TCP) is one of these techniques to improving regression testing. TCP reorders test cases according to some criterion so that test case with higher priority can be executed earlier. One goal of TCP is to detect faults faster and terminate testing process early with much less overhead. Research shows that TCP can help tester improve the rate of fault detection and the developers can start debugging work early [6, 7]. To date, various prioritization techniques have been proposed and empirically studied [3, 6-8]. These techniques analyze source codes and historical execution data to obtain useful information, such as code coverage of test cases, code complexity, test costs, fault severities, in the form of alone or mix and then prioritize these test cases [6, 9-11].

In the evolution of software products, program codes are changed constantly. Most of the new faults are introduced in the program by these changes, and we need to verify the modified program to ensure its validity. To do this, developers need to execute test cases in a new order to cover these changed codes earlier. However, Hao et al. [29] indicate that the TCP technique which purses the optimality by taking coverage as an intermediate goal cannot be more effective than the greedy technique in terms of fault detection. Meanwhile, they also prove that an ideal technique which orders test cases by their fault detection capabilities outperforms the coverage-based techniques. Therefore, in this paper, we propose our TCP algorithm based on the fault detection capabilities of test cases.

It is generally believed that a test case covering more modified codes is considered to have a higher capability of revealing faults than other test cases, which is based on the assumption that the modified codes have the same possibility of being faulty. However, the possibilities of being faulty of the modified codes are not always the same. Calculating the fault detection capabilities of test cases just based on the codes they cover is not precise. If we can evaluate the faulty possibility of each modified code, the fault detection capability of each test case may be calculated more accurately.

In this paper, we propose a test case prioritization algorithm based on program changes and method (function) invocation relationship (PC-IR). This algorithm is discussed at method-level (function-level, and 'function' will be omitted below.). By estimating the risk value of each method through static source-code analysis, we can calculate the fault detection capabilities of test cases with their dynamic execution information, and finally prioritize test cases with their fault detection capabilities by using an ILP model. Here, the risk of a method represents the possibility of the occurrence of the faults which are contained in this method. Higher risk value means greater probability of the method that contains faults and more easily the faults can be detected. It is influenced by the modified degree and the invoked degree of a method, and

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.23

through static source-code analysis, we can obtain the above two information. Through dynamic program execution, we can obtain the coverage information of each test case on the methods, and different coverage data means different capabilities of revealing faults of test cases. We can calculate the fault detection capability of each test case based on the risk values of methods that are covered by this test case. In addition, previous research [33] demonstrates that the ILP method is effective for selecting a subset from a test suite, and test case prioritization can be regarded as the process of continuously selecting test cases for structuring the subset of the test suite. Therefore, we use ILP for prioritizing test cases according to fault detection capabilities of test cases in our algorithm.

To investigate the effectiveness of our algorithm, we have designed and conducted empirical studies on 9 C programs and 2 Java programs. We compare our proposed algorithm with four other TCP approaches which are popularly used to benchmark the performance of other prioritization approaches. Experimental results show that our proposed approach can effectively improve the effectiveness of regression testing.

II. Background and Related Work

To improve the software testing activity in regression testing, researchers have proposed many metrics as well as techniques for TCP in recent years. The coverage-based TCP technique is one of the most widely studied techniques [6][8-10] [13]. Most of these techniques usually sort test cases to maximum coverage through the search algorithm. Total greedy algorithm [6] and additional greedy algorithm [10] are two basic code-coverage algorithms for TCP, which are generic for different coverage criteria. They are also popularly as benchmarks to measure other TCP techniques. In addition to these two strategies, researchers have also investigated other generic strategies. Hao et al. [8] propose two models that unify the total and additional algorithms. Their empirical results prove that the additional greedy algorithm is the most effective TCP approach in terms of fault detection capability. Li et al. [9] provides a comprehensive overview of the 2-optimal greedy strategy, a hill-climbing strategy, and a genetic programming strategy. Jiang et al. [14] propose adaptive random prioritization (ART). Carlson et al. [15] implement new prioritization techniques that incorporate a clustering approach and utilize code coverage, code complexity, and history data on real faults that gathered from previous test process. Xia et al.[16] employ trend analysis to give priorities to test cases that can quickly increase the diversity of suspiciousness scores generated by fault localization techniques for various program elements. Wang et al. [17] propose a global similarity-based regression test case prioritization approach based on the distance between pair-wise test cases.

Except for the above strategies using code coverage information, *program change* is also used to help prioritize test cases. Sherriff et al. [18] utilize change records to gather change impact information and prioritize test cases accordingly. Saha et al. [19] transform regression test prioritization to a standard Information Retrieval problem such that the differences between two program versions form the query and the tests constitute the document collection. Other types of information are also for TCP, such as historical testing information [12][20-21], relevant slice of testing output [22], testing requirement data [23-24], testing input data [25].

In addition, ILP model is also used to solve TCP problem. At first, researchers propose to use ILP model to solve the problem of TCP with time or resource constraints. Zhang et al. [26] fully consider the time overhead in regression testing, and propose a novel time-aware TCP approach using ILP. They perform experiments on four ILP techniques based on two basic greedy algorithms (total greedy algorithm and additional greedy algorithm) and other 7 techniques. Empirical results show that the two additional greedy techniques based on ILP is superior to other approaches. Jabbarvand et al. [27] present an energy-aware test-suite minimization approach to reduce the number of tests needed to effectively test the energy properties of an Android app. This approach relies on an energy-aware coverage criterion to calculate the coverage that a test case can achieve, and leverages integer programming to model the minimization problem. Hao et al. [28] formulate the optimal coverage-based TCP as an ILP problem to obtain an optimal test case order, and empirical studies show that this optimal technique does not outperform the additional coverage-based technique in terms of either fault detection or execution time. However, the optimal fault-detected-based technique, an ideal technique, outperforms the two coverage-based techniques in terms of fault detection significantly.

As shown above, there are numerous TCP techniques using various types of information. Our algorithm uses program changes, direct method invocation relationships and coverage information to estimate the fault detection capability of each test case, and then use ILP for prioritizing all test cases. Empirical study in [28] has demonstrated that coverage-based TCP technique which takes program elements coverage as the intermediate goal cannot outperform additional greedy technique in terms of a fault detection or execution time. This is because that covering a program element may not guarantee the detection of faults in the element. Therefore, instead of program elements coverage, our algorithm takes the fault detection capabilities of test cases as the goal to prioritizing test cases. Our algorithm essentially tends to be the ideal but impractical fault-detected-based technique which prioritizes test cases based on their number of detected faults.

III. Test Case Prioritization ALGORITHM

We now introduce our prioritization approach in this section. PC-IR contains two main steps: (1) Calculation of fault detection capability of test case; (2) Test cases prioritization process. The following subsections describe each of these steps in detail.

A. Definition

In order to clarify the prioritization idea in this paper, we redefine the test case prioritization problem based on the definition of test case prioritization by Rothermel et al. [6] as follows.

Given: A new program version P consist of q program elements $M = \{m_1,m_2,...,m_q\}$; A test suite $T = \{t_1,t_2,...,t_n\}$, PT is the set of all possible permutations of T; $w(t_i)$ $(1 \le i \le n)$, a non-negative function that represents the fault detection capability of each test case.

Problem: Find $T' \in PT$ such that for every other T'' ($T'' \in PT$) ($T'' \neq T'$)

$$\left[\sum_{i=1}^{k} {}_{t_i \in T'} w(t_i) \geq \sum_{i=1}^{k} {}_{t_i \in T''} w(t_i) \quad (1 \leq k \leq n) \right]$$

In this definition, program element denotes a unique unit in a program, which can be defined in different grains, fine-grain (e.g., statements) and coarse-grain (e.g., methods or classes), and in our approach, we define program element coarse-grained as methods. T' represents the optimal permutation of T; $w(t_i)$ can be calculated by using the risk values of methods which are covered by test case t_i. The greater $w(t_i)$ is, the stronger the fault detection capability of t_i. The object function denotes that the sum of the fault detection capability of the first k test cases in T' is always greater than that of the first k test cases in T''.

B. Test Case Prioritization Overview

PC-IR contains two main steps: (1) Calculation of fault detection capability of test case, which outputs the $w(t_i)$ of each test case; (2) Test cases prioritization, which provides a permutation T' of the original test suite T for effective testing according to the sorting criterion.

In the first step, $w(t_i)$ can be calculated by using the risk values of methods and method-coverage data of t_i. The risk of a method m_k ($1 \leq k \leq q$) represents not only the probability of a method that contains faults, but also the possibility of the faults occurrence. It is influenced by the following two factors: (1) the degree of m_k's modification, and (2) the number of methods which invoke m_k in the program. The degree of m_k's modification can be represented by the number of modified code lines. Empirical studies in [32] show that "the likelihood of a vulnerability increases with the number of lines changed". Therefore, the greater the number of the modified lines in a method, the greater the likelihood of the method that contains faults. The invoked degree of a method can be obtained from the method call graph, and it represents the degree of difficulty that faults in the method are revealed. If m_k is invoked by a number of other methods in the program, it may frequently be executed during execution of a test case and faults in this method will occur easily. A method m_k which has a great modification degree and is invoked by a large number of other methods will have a high risk value. A test case covering methods like m_k will contribute more information for revealing faults when compares with the ones that cover less methods like m_k. Meanwhile, if the coverage percentage of a test case is high, we believe that there is a desirable probability that this test case can cover the changed codes, that is, this test case is more likely to cover the faulty codes. We hold that test cases like this may have strong fault detection capabilities.

In the process of test case prioritization, we use ILP to solve this prioritization problem according to fault detection capability. In this process, with test cases being selected constantly, the remaining test cases which cover the same methods with already selected test cases, their contribution to detect new faults decrease significantly. Therefore, we adjust $w(t_i)$ of the remaining test cases by analyzing the coverage information of the test cases that have been sorted.

Different from the prioritization techniques which only depend on code coverage, PC-IR combines the coverage information of test cases and the risk value of each method itself, and highlights the additional $w(t_i)$ of each test case that is going to be selected. In the prioritization process, PC-IR prefers to select the test case whose $w(t_i)$ is higher rather than select a test case covers most methods that have not been covered by previously selected test cases. Therefore, PC-IR ensures that the prioritized test cases can quickly cover all high suspicious methods. Furthermore, in the prioritization process of PC-IR, if all the methods have been covered by at least one prioritized test case or there is no test case that can cover new method, PC-IR will reselect a test case with the maximum fault detection capability from the remaining test cases by assuming that no test cases have been selected, and cycle the prioritization process until all the test cases are reordered. More details will be described below.

C. Calculation of Fault Detection Capability

Due to the different contribution of test cases to fault detection, a corresponding priority can be assigned to each test case according to the contribution values of test cases. In previous literature [26] [28], each program element is given the same probability of being faulty, and the testing priority of each test case only depends on its coverage on the program elements. Test case which can cover more program elements, its initial priority will be higher (The final execution priority may change with the selecting of test cases). However, the faulty probability of each program element is different, and it is not accurate that sorting test cases only depends on the amount of code coverage of test cases. PC-IR not only considers the coverage of test cases on methods, but also evaluates the risk value of each method itself, and using an estimated fault detection capability of each test case measures its contribution to fault detection.

The fault detection information of test cases is an important kind of information which can directly reflect the capability of revealing faults of a test case, and it can directly affect the position of the test case in the sorting list. Therefore, using the detected faults by a test case for prioritization will always obtain a better testing effectiveness. However, it is unknown which faults each test case can detect, using the detected faults of test cases for prioritization is not applicable in practice. PC-IR uses an estimated fault detection capability of the test case to prioritizing test cases instead of using the actual detected faults of test cases.

In order to provide a reliable sequence, it is necessary to estimate the fault detection capabilities of test cases as accurately as possible. In this paper, the fault detection capability of a test case is defined as the sum of risk values of all program methods that the test case covers. In previous section, we have introduced the definition of risk value of the program method and known that higher risk value means greater probability of the method that contains faults and more easily the faults can be detected. The risk value of a method is influenced by the degree of its modification and the number of methods which invoke it in the program. Through static source-code analysis, we can collect these two kinds of information. Since the coverage of test cases on the new program version is unknown, through collecting the execution traces of test cases on a previous version, we can obtain the

historical method coverage of test cases. Finally, we calculate the fault detection capability of each test case based on the risk values of methods that are covered by this test case.

Now we discuss how to perform risk assessment for a program method m_k ($1 \le k \le q$). Through static source-code analysis, we can identify the two risk factors: (1) the degree of m_k's modification; (2) the number of methods that invoke m_k. The degree of m_k's modification can be obtained by comparing the differences between two program versions, and it is measured through lines of modified codes (mLOC). Here, codes modification includes existing codes change, deletion, and new codes addition. Methods with higher modified LOC numbers tend to contain more faults. Method invocation relationship reflects the structural dependency of the program, which can be obtained from method call graph. If a method is invoked by a number of methods in the program, it may frequently be executed during execution of a test case and faults in this method will easily occur. Since method coverage information cannot precisely reflect the frequency of the method that is executed during execution of a test case, we believe that, in the program, the more methods that m_k is invoked by, the easier the faults in m_k can be detected. Taking into account the above two factors, the risk value of each method which represents the probability of the fault occurrence in this method can be calculated as:

$$r(m_k) = mLOC_k \times d_k \tag{1}$$

where $mLOC_k$ represents the changed code lines in method m_k, d_k represents the number of methods that invoke method m_k in the program. To prevent the omission of faults in the program, for the methods in which codes are not changed, we set 1 to its $mLOC_k$. And for the method which is not invoked by any other methods, we set its $d_k = 1$.

As discussed before, the $w(t_i)$ of a test case t_i can be measured by the sum of the risk values of the methods which are covered by t_i. If a test case covers a number of methods with high risk values, we believe that it is more capable of revealing faults. Since the execution traces of test cases on the current program version cannot be obtained in advance, we will use the historical execution traces to estimate the fault detection capability of each test case.

For a test case, even if it can cover a method, we still cannot guarantee that this test case can cover the changed codes, and faults in this method may not be detected by this test case. Therefore, we use the percentage of code coverage in method when executing test case t_i as the probability that the faults in method m_k can be detected by test case t_i. If the percentage of code coverage of test case t_i on previous version is p_i, the capability of t_i that can detect faults in method m_k is

$$R_{ik} = r(m_k) \times p_i \tag{2}$$

where $r(m_k)$ represents the risk value of method m_k. Note that Formula 2 is based on the assumption that t_i covers method m_k, and if t_i does not cover m_k, t_i cannot detect any faults in m_k. Therefore, we should first obtain the coverage information of t_i on each method. Formula 3 defines a Boolean variable c_{ik} ($1 \le i \le n$ and $1 \le k \le q$) to represent whether t_i covers m_k,

$$c_{ik} = \begin{cases} 1, & \text{if } t_i \text{ covers } m_k \\ 0, & \text{otherwise} \end{cases} \quad (1 \le i \le n \text{ and } 1 \le k \le q) \tag{3}$$

If a method m_k ($1 \le k \le q$) is covered by a test case, we assign 1 to this method in the coverage vector of this test case; otherwise, 0 is assigned. Therefore, the fault detection capability of each test case can be calculated as follows.

$$w(t_i) = \frac{\sum_{k=1}^{q} R_{ik} \times c_{ik}}{\sum_{k=1}^{q} max\{R_{1k}, R_{2k}, ..., R_{nk}\}} \quad (1 \le i \le n) \tag{4}$$

where q is the number of methods in the program, n is the number of test cases. $w(t_i)$ takes a value between 0 and 1. A test case with a higher $w(t_i)$ is more likely to detect faults.

Different from the techniques which assign ranking priorities for test cases only depend on the coverage information, our algorithm considers the faulty possibility of each program method and estimates the fault detection capability of each test case combining with the coverage information, and reorder test cases based on their fault detection capabilities. When using PC-IR, even if a test case covers more methods, its fault detection capability may be lower than those test cases which cover less methods but whose risk are all great.

D. Test Case Prioritization

In this section, we use ILP to solve the prioritization problem and complete the prioritization of test cases.

(1) Decision Variables

Similar to the research in [28] which use the ILP to model the prioritization problem, we also define two decision variables: the position variable of test case in the prioritized order T' (T' is the optimal permutation of T) in Formula 5, and the method coverage variable obtained from the selected test cases in Formula 6. In Formula 5, we use a $n*n$ Boolean variable, o_{ij} ($1 \le i, j \le n$), to represent whether the j-th position in T' is test case t_i. cov_{jk} ($1 \le j \le n$ and $1 \le k \le m$) in Formula 6 is a $n*m$ Boolean variable, which denotes whether method m_k is covered by at least one test case in the first j test cases of T'.

$$o_{ij} = \begin{cases} 1, & \text{if the } j\text{-th test case in } T' \text{ is } t_i \\ 0, & \text{otherwise} \end{cases} \quad (1 \le i, j \le n) \tag{5}$$

$$cov_{jk} = \begin{cases} 1, & \text{if the first } j \text{ test cases cover } m_k \\ & (1 \le j \le n \text{ and } 1 \le k \le m) \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

(2) Objective Function

When prioritizing test cases, we give higher priorities to test cases with higher $w(t_i)$ values. Therefore, the criteria of prioritization should be defined to select the test case which has the maximal $w(t_i)$ value among the remaining test cases. That is to say, when a test case is selected into T', the sum of the fault

detection capability of test cases in the new sequence will not be less than the fault detection capability of any other sequence which select other test case into *T'*. Overall, our objective function can be defined as:

$$maximize \sum_{j=1}^{n}\sum_{i=1}^{n} w(t_i)\times o_{ij}$$
$$\Rightarrow maximize \sum_{j=1}^{n} \frac{\sum_{i=1}^{n}\sum_{k=1}^{q} R_{ik}\times c_{ik}\times o_{ij}}{\sum_{k=1}^{q} max\{R_{1k}, R_{2k},..., R_{nk}\}} \quad (7)$$

In Formula 7, $o_{ij}\times c_{ik}$ takes a value of 1, which represents that the test case t_i is in the *j*-th position of *T'* and covers the method m_k, and 0 represents test case in the *j*-th position is not t_i, or t_i does not cover the method m_k.

As we know, additional greedy algorithm is widely considered as the most effective TCP approach to quickly expose faults from a program[8]. This algorithm reorders test cases by selecting the next test case that covers more statements that have not been covered by previously selected test cases. Similarly, with test cases being selected constantly, the remaining test cases which cover the same methods with already selected test cases, their contribution to detect new faults decrease significantly. That is, a test case that covers method already covered by previously prioritized test cases is not important anymore to detect new faults. Therefore, in our prioritization process, we will weaken the $w(t_i)$ values of the remaining test cases which cover same methods with the test cases that have been selected. Assuming that *j*-1 test cases have been sorted, the $w(t_i)$ values of the remaining test cases should be adjusted as follows.

$$w(t_i) = \frac{\sum_{k=1}^{q} R_{ik}\times c_{ik}\times\left(1-cov_{j-1,k}\right)}{\sum_{k=1}^{q} max\{R_{1k}, R_{2k},..., R_{nk}\}} \quad \left(\forall t_i \in \{T - T_{j-1}\},\ 2\le j\le n\right) \quad (8)$$

where the coefficient $(1\text{-}cov_{j-1,k})$ represents the remaining methods which have not been covered by the first *j*-1 test cases, and '1' represents that m_k has not been covered by the first *j*-1 test cases, and '0' otherwise. T_{j-1} is the set of test cases which have been selected, that is, the first *j*-1 test cases in *T'*. Meanwhile, the objective function is adjusted as follows.

$$maximize \sum_{j=2}^{n} \frac{\sum_{i=1}^{n}\sum_{k=1}^{q} R_{ik}\times c_{ik}\times\left(1-cov_{j-1,k}\right)\times o_{ij}}{\sum_{k=1}^{q} max\{R_{1k}, R_{2k},..., R_{nk}\}} \quad (9)$$

The coefficient $c_{ik}\times(1\text{-}cov_{j-1,k})\times o_{ij}$ takes a value of 1, if the *j*-th test case covers method m_k that is not covered by previously selected *j*-1 test cases, and 0 otherwise. That is, $c_{ik}\times(1\text{-}cov_{j-1,k})\times o_{ij}$ denotes the new methods covered by the *j*-th test case that are not covered by the first *j* test cases. Therefore, $c_{ik}\times(1\text{-}cov_{j-1,k})\times o_{ij}$ can be replaced by the coefficient $(cov_{jk} - cov_{j-1,k})$. We use R_{jk}' to represent the capability of the *j*-th test case in *T'* that can reveal faults in m_k. The objective function can be rewritten as follows.

$$maximize \sum_{j=2}^{n} \frac{\sum_{k=1}^{q} R'_{jk}\times\left(cov_{jk} - cov_{j-1,k}\right)}{\sum_{k=1}^{q} max\{R_{1k}, R_{2k},..., R_{nk}\}} \quad (10)$$

(3) Constraints

To certify that there is one and only one test case in each position of the permutation of *T*, and simultaneously each test case appears one and only once in the permutation of *T*, there are two constraints for the values of o_{ij} in the ILP model as follows.

$$\sum_{i=1}^{n} o_{ij} = 1 \quad (1\le j\le n) \quad (11)$$

$$\sum_{j=1}^{n} o_{ij} = 1 \quad (1\le i\le n) \quad (12)$$

With the increase of selected test cases, there are some relationships between the methods covered by the first *j*-1 test cases and the methods covered by the *j*-th test case.

$$\sum_{i=1}^{n} c_{ik}\times o_{ij} \le cov_{jk} \quad (1\le k\le q, 1\le j\le n) \quad (13)$$

$$cov_{jk} \ge cov_{j-1,k} \quad (1\le k\le q, 2\le j\le n) \quad (14)$$

$$cov_{jk} - cov_{j-1,k} \le \sum_{i=1}^{n} c_{ik}\times o_{ij} \quad (1\le k\le q, 2\le j\le n) \quad (15)$$

In Formula 13, $\sum_{i=1}^{n} c_{ik}\times o_{ij}$ $(1\le k\le q)$ represents whether the *j*-th test case in *T'* covers the method m_k. This constraint ensures that the methods covered by the *j*-th test cases must be covered by the first *j* test cases in *T'*. Specifically, when *j* = 1, $\sum_{i=1}^{n} c_{ik}\times o_{i1} = cov_{1k}\ (1\le k\le q)$. Formula 14 ensures that the number of methods covered by the first *j* test cases must contain the methods covered by the first *j*-1 test cases. The constraint in Formula 15 certifies that the new covered methods by the *j*-th test case must be included in the method coverage of the test case in the *j*-th position. Formula 13, 14, 15 illustrate the relationships between the position and method coverage of test cases in *T'*.

In the prioritized sequence obtained by using the coverage-based additional greedy technique, we can sure that the *j*-th test case has a higher priority than the (*j*+1)-th test case because the *j*-th test case can cover more methods which are not covered by the first *j*-1 test cases than that of the (*j*+1)-th test case. However, in *T'* obtained by using our prioritization algorithm, we cannot certain that which test case in the (*j*+1)-th position and *j*-th position will cover more methods that are not covered

by the first j-1test cases. That is, we cannot define the relationship between $\sum_{i=1}^{n}\sum_{k=1}^{q} c_{ik}\times o_{ij}\times\left(1\text{-}cov_{j\text{-}1,k}\right)$ and $\sum_{i=1}^{n}\sum_{k=1}^{q} c_{ik}\times o_{i,j+1}\times\left(1-cov_{j\text{-}1,k}\right)$ $(2 \le j \le n)$. This is because our algorithm prefers to select the test case whose fault detection capability is higher rather than select a test case covers most methods that have not been covered by previously selected test cases. But in T', we can sure that the j-th test case has a higher priority than the (j+1)-th test case because the j-th test case has a higher capability of detecting new faults than the (j+1)-th test case. Formula 16 certifies that the j-th test case has a higher priority than the (j+1)-th test case.

$$\sum_{i=1}^{n}\sum_{k=1}^{q} R_{ik}\times c_{ik}\times o_{ij}\times\left(1-cov_{j-1,k}\right)\ge \sum_{i=1}^{n}\sum_{k=1}^{q} R_{ik}\times c_{ik}\times o_{i,j+1}\times\left(1-cov_{j-1,k}\right) \quad (2\le j\le n-1) \tag{16}$$

IV. EXPERIMENTS

A. Research Questions

To evaluate the effectiveness of PC-IR for fault detection, we design and implement a set of empirical studies to answer the research question: Is PC-IR effective when compared with the efficient prioritization techniques in terms of APFD (Average Percentage of Fault Detection) metric?

B. Subject Programs

Our experiments are conducted by using 9 C programs and 2 Java programs. We obtain the first 9 programs from SIR http://sir.unl.edu, and download Apache Commons Lang from its host website. Since our algorithm needs to use the historical coverage data of previous version, we consider a number of consecutive versions for each program. As these subject programs have only a small number of manually seeded faults available, we generated faulty versions for each program by using different approaches. For Siemens programs, the method of generating each faulty version is similar to the method used by Elbaum et al. [10]. SIR provides a number of faulty versions of Siemens programs that contain exactly one fault. All these versions are independent of each other, and faults in them may belong to different fault types. We merge various types of faults into the source version to generate a multi-fault version. For the two Unix programs and two Java programs, we use the Proteum [29] and MuJave[30] to generate the faulty versions, respectively. Proteum and MuJava are the mutation testing tools which are used to assess the sufficiency of testing on C programs and Java programs. Mutants can be used as the substitute for program faults in the study of test case prioritization techniques [28][30]. However, mutants which cannot be killed by any test cases, or can be easily killed (can be killed by more than 10% of all test cases), or have multiple exactly equivalent mutants are inappropriate in our experiments. We discard the first two kinds of mutants, and randomly keep one from each equivalent mutant set. Meanwhile, due to code changes, some test cases cannot be executed successfully, we exclude these test cases and generate the corresponding test suite for each program version. Table 1 shows the details of the subject programs used in our experiments.

TABLE1 DESCRIPTION OF SUBJECT PROGRAMS

Program	LOC	Methods	Test Cases	Versions
print_tokens	695 ~ 743	18	4071	6
print_tokens2	562 ~ 581	19	4031	8
replace	563 ~ 566	21	5542	15
schedule	360 ~ 369	18	2650	5
schedule2	329 ~335	16	2710	5
tcas	162 ~ 176	9	1608	8
totinfo	549 ~ 560	7	1052	9
flex	10054 ~ 12407	162	567	5
sed	14138 ~ 14427	255	370	5
xml-security	16632 ~ 17154	1231	97	3
commons lang	52288 ~ 55531	2151	1874	5

C. Evaluation Metrics

In order to evaluate the effectiveness of the test case prioritization techniques, Rothermel et al. [6] proposed the Average Percentage of Fault Detected (APFD) metric that is widely used in test case prioritization research. This metric measures the effectiveness of prioritization techniques in terms of the capacity of fault detection of a test suite, and is defined as follow.

$$APFD = 1-\frac{TF_1+TF_2+\ldots+TF_m}{nm}+\frac{1}{2n} \tag{17}$$

where n is the total number of test cases, m is the number of faults in program, TF_k represents the position of the first test case in prioritized order of T which reveals fault k. APFD is a measure of fault detection rate of a test suite and its value ranges from 0 to 1. Higher APFD values indicate more effectiveness of the prioritized test suite. It should be noted that the definition of APFD is based on the assumption that all test case costs are uniform and all fault severities are uniform. In our research, we use APFD as the evaluation metrics for the test case prioritization.

D. Experimental Process

In order to perform our prioritization approach PC-IR, we need three data sets: the static call graph of program methods, code difference between two sequential program versions, and code coverage information of the original test suite on previous versions.

The first two kinds of data can be obtained from static analysis. The static call graph can be obtained by analyzing program source code without executing the program. In this paper, to statically analyze the subject programs, we use CodeViz[34] and WALA framework [35] to extract the call graphs for C and Java programs, respectively. We search the program differences between two versions at the line level by applying UNIX *diff* recursively while ignoring spaces and blank lines, and record the number of changed lines in each method. Using the above two kinds of information, we can preliminarily calculate the risk value of each method that contains faults. To collect the code coverage data of test cases executed over its test pool, we use some instrumentation tool such as "gcov" command of GNU C compiler for C programs

and EMMA [36] for Java programs. Since the codes in each test version are changed differently, the codes that were covered by even same test cases in different test versions are not same. In theory, the test version whose code change is similar to the current test version, its code coverage can be used as the historical information in current testing version.

To evaluate the performance of our proposed algorithm, random prioritization, additional greedy prioritization algorithm, and the ideal optimal technique [28] are also used for comparison in our experiments. Random algorithm orders test cases just one by one randomly. Since random approach is based on random selection, and it could generate different test case orders, we repeat 20 times for each program version to obtain averages that can portray typical performance. Additional greedy algorithm is widely considered as the most effective TCP approach to quickly expose faults from a program [8], and it reorders test cases by selecting next test case that covers more program elements that have not been covered by previously selected test cases. Although PC-IR is based on the method coverage, for better comparing with other TCP techniques, we use two coverage criteria (statement coverage and method coverage) when performing additional technique in our experiments. The ideal optimal technique prioritized test cases based on their number of detected faults that have been not detected by existing selected test cases. Although the ideal technique is not practical, we use it to study the disparity between the effectiveness of PC-IR and the upper bound of TCP. Therefore, our experiments use the following five prioritization methods: random technique (Random), additional greedy technique based on statement coverage (AG-S), additional greedy technique based on method coverage technique based on method coverage (AG-M), ideal optimal technique (Optimal), and PC-IR.

All experimental studies in this paper are conducted on an Intel Core i2 CPU Q8400 @ 2.66 GHz, 4.00GB RAM, with Ubuntu 11.10.

E. Experimental results analysis

Our research question considers whether PC-IR is effective, therefore, we compare it with some of the best TCP techniques. Table 2 shows the APFD values of the experimental results of each approach for all subject programs, respectively. In Table 2, the 'Approach' column shows the prioritization techniques, the 'Mean' column shows the average APFD values of all versions with each prioritization approach, and the 'Improvement' column shows the average improvement APFD of PC-IR over other approaches. For example, the mean APFD value of Random technique for *print_tokens* is 73.71%, while the APFD value produced by PC-IR is 91.75%. The improvement APFD value of PC-IR over Random technique is 18.04%.

Let's analyze the APFD values of the prioritization approaches in the experiment in detail. As seen from Table 2, the average APFD values of PC-IR are greater than Random, AG-S, and AG-M techniques on most of programs, except for *schedule* and *xml_security* programs. It means that PC-IR outperforms other approaches across majority of program versions while does not consider test case costs and fault severities. Comparing with Random technique, PC-IR has a significant improvement on mean APFD (improvement of more than 15% on most of programs). PC-IR also outperforms AG-M approach over most of programs, even on program *xml_security*, there is no significant difference between their APFD values. When compared with the most effective AG-S prioritization approach, PC-IR also performs better on 9 of 11 programs, and improvements are significant on 4 programs (improve more than 3%). Moreover, from Table 2, we also observe that the average APFD values of AG-S are greater than that of AG-M. This is because it is easier to produce a prioritized test sequence with high coarse-granularity, but a test case can cover a method does not mean that it can cover the faulty statements in this method. Although PC-IR is proposed on high coarse-granularity (method or function), the prioritization criteria is the fault detection capability of each test case. Therefore, PC-IR will be more effective than the TCP approach only based on high coarse-granularity coverage. Results from Table 2 also verify this conclusion. Further, to show our results visually, boxplots for each technique for programs with their APFD values are shown in Fig. 1. The figure contains 11 subfigures, and each subfigure shows the distribution of APFD values of all four prioritization approaches for a program. The horizontal axis corresponds to TCP approaches, and the vertical axis corresponds to APFD values. From Fig. 1, we observe that PC-IR perform outstandingly. We observe that PC-IR is significantly more effective than random technique and AG-M technique (except on *xml_security*). Moreover, except for the case on *schedule* and *xml_security*, we find that the median APFD of PC-IR is visually more effective than AG-S approach. Examining the boxplots for each approach in Fig. 1, the trends observed from Table 2 (average values) are consistent with the results shown from the boxplots. In particular, for all four approaches, the differences between the best and worst APFD values are noticeable.

To compare our approach with the two more effective TCP approaches (AG-S and AG-M) more intuitively and in more detail, the bar charts of APFD values on each faulty version are given in Fig. 2. (Taking into account the size of program and the average APFD of each program, we take *print_tokens, schedule*, *sed,* and *xml_security* as examples in Fig.2.). From Fig. 2, we observe that PC-IR performs better than the other two approaches on most of the faulty versions in terms of APFD values, although in some versions, the improvement of effectiveness is not significant.

The goal of test case prioritization is to use fewer test cases to detect more faults in the program. Therefore, we have studied the relationship between the percentage of detected faults and the percentage of executed test cases on each faulty version. Considering the sizes of programs, the number of test cases, and the number of faults contained in each program version, we select the following versions as example for investigating this relationship, i.e. version 1 and 3 of *print_tokens* program (small-sized program, contain 3 and 10 faults, respectively, and almost 4000 test cases), version 6 of *replace* (small-sized program, contains 14 faults, and more than 5000 test cases), version 3 of *totinfo*(small-sized program, contains 9 faults, and relative few test cases), version 2 of *flex* (middle-sized program, contains 21 faults, and more than 500 test cases), version 1 of *sed* (middle-sized program, contains 10 faults, and 370 test cases), version 3 of *commons lang* (middle-sized program, contains 17 faults, and more than 1800 test cases), and version 1 of *xml_security* (middle-sized program, contains 7 faults, and only 97 test cases). Fig.3

shows the results of this relationship for the four TCP approaches. From Fig. 3, we can observe that our approach can always detect more faults when executing the same percentage of test cases. For example, when 10% test cases have been executed on version 3 of *print_tokens* program, the four TCP approaches (Random, AG-S, AG-M, and PC-IR) can detect 40%, 70%, 60%, and 80% faults, respectively. This result also proves that our approach can improve the rate of fault detection.

TABLE 2 APFD VALUES OF THE PRIORITIZATION APPROACHES FOR TESTED PROGRAMS

Program	Approach	Mean	Improvem	Program	Approach	Mean	Improvem	Program	Approach	Mean	Improvem
print_tokens	Radom	73.71	18.04	print_tokens2	Radom	69.34	20.59	replace	Radom	71.52	14.41
	AG-S	89.8	1.95		AG-S	86.87	3.06		AG-S	85.13	0.8
	AG-M	86.83	4.92		AG-M	83.06	6.87		AG-M	84.8	1.13
	Optimal	99.86	-8.11		Optimal	99.74	-9.81		Optimal	99.92	-13.99
	PC-IR	91.75	--		PC-IR	89.93	--		PC-IR	85.93	--
schedule	Radom	72.5	13.94	schedule2	Radom	70.37	22.73	tcas	Radom	53.26	21.8
	AG-S	87.08	-0.64		AG-S	90.22	2.88		AG-S	73.2	1.86
	AG-M	86.17	0.27		AG-M	88.46	4.64		AG-M	71.52	3.54
	Optimal	99.25	-12.81		Optimal	99.92	-6.82		Optimal	99.86	-24.8
	PC-IR	86.44	--		PC-IR	93.1	--		PC-IR	75.06	--
totinfo	Radom	71.96	20.77	flex	Radom	74.68	14.72	sed	Radom	68.24	24.36
	AG-S	89.35	3.38		AG-S	87.66	1.74		AG-S	88.18	4.42
	AG-M	89.04	3.69		AG-M	84.59	4.81		AG-M	85.04	7.56
	Optimal	99.77	-7.04		Optimal	99.71	-10.31		Optimal	99.45	-6.85
	PC-IR	92.73	--		PC-IR	89.4	--		PC-IR	92.6	--
xml_security	Radom	73.33	6.69	commons lang	Radom	75.18	17.45				
	AG-S	81.6	-1.58		AG-S	89.39	3.24				
	AG-M	80.67	-0.65		AG-M	88.97	3.66				
	Optimal	97.94	-17.92		Optimal	99.79	-7.16				
	PC-IR	80.02	--		PC-IR	92.63	--				

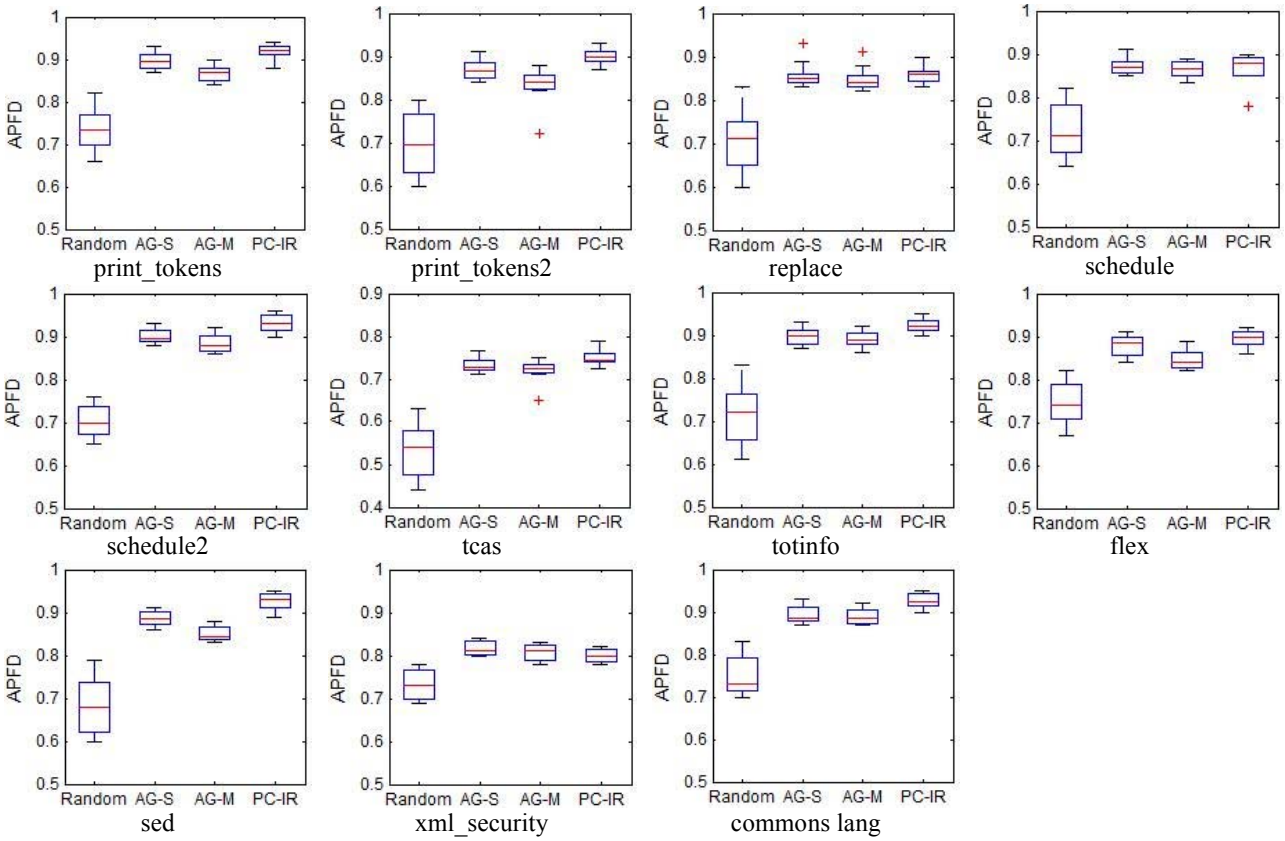

Figure 1. APFD distributions for all subject programs

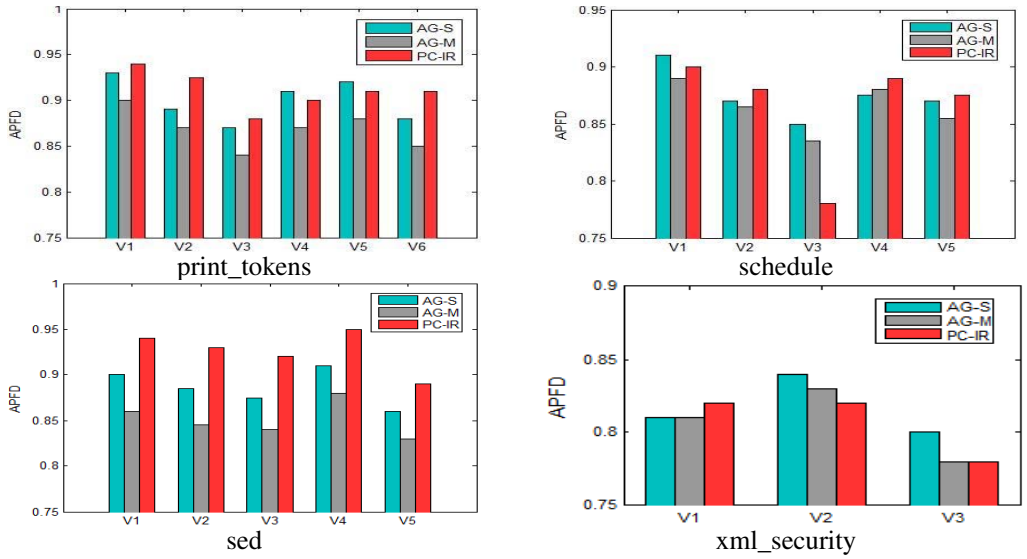

Figure 2. APFD of AG-S, AG-M and PC-IR approaches on each faulty version

V. CONCLUSION

In this paper, we propose a new test case prioritization algorithm based on program changes and method invocation relationship. Firstly, this algorithm estimates the risk value of each program method through static source-code analysis. By combining with the historical dynamic execution information of test cases on methods, fault detection capability of each test case can be effectively calculated. Finally, test cases are prioritized by reducing the TCP to an ILP problem. Our experiments are conducted on 11 subject programs. The results have shown that our proposed approach is always superior to, or at least as effective as, some of the well accepted prioritization approaches in the literature, including two additional greedy algorithms based on statement and method coverage respectively. In future work, we will further investigate how to improve the effectiveness of regression testing by incorporating historical information, fault severity, time overhead etc. We will also empirically evaluate it on more benchmarks.

ACKNOWLEDGMENTS

This work was partially supported by the NSF of China under grants No. 61702334 and No. 61772200, Shanghai Pujiang Talent Program under grants No. 17PJ1401900. Shanghai Municipal Natural Science Foundation under Grants No.17ZR1406900 and 17ZR1429700. Educational Research Fund of ECUST under Grant No.ZH1726108. The Collaborative Innovation Foundation of Shanghai Institute of Technology under Grants No. XTCX2016-20.

REFERENCES

[1] T.L. Graves, M.J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques", ACM Transactions on Software Engineering & Methodology, vol.10, no.2, pp.184–208, 2001

[2] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," IEEE Transactions on Software Engineering, vol.33, no.2, pp.108-123, 2007.

[3] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," IEEE Transactions on Software Engineering, vol.38, no.6, pp.1258-1275,2012

[4] Y. Yu, J.A. Jones, and M.J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," 30th International Conference on Software Engineering, pp.201-210, 2008.

[5] D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, "Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study," 21th IEEE International Conference on Software Maintenance (ICSM), pp.683-686, 2005.

[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] T. Xie, L. Zhang, X. Xiao, Y.F. Xiong, and D. Hao, "Cooperative software testing and analysis: advances and challenges," Journal of Computer Science & Technology, vol. 29, no.4, pp.713-723, 2014.

[8] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," ACM Transactions on Software Engineering & Methodology, vol. 24, no.2, pp.1-31, 2014.

[9] 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, 2007.

[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] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," 14th International Symposium on Software Reliability Engineering, pp.17-20, 2003.

[12] J.M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," 24th International Conference on Software Engineering , pp.119-129, 2002.

[13] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," ACM SIGSOFT Software Engineering Notes, vol.27, no.4, pp.97-106, 2002.

[14] B. Jiang, Z. Zhang, W.K. Chan, and T.H. Tse, "Adaptive random test case prioritization," IEEE/ACM International Conference on Automated Software Engineering, pp.233-244, 2009.

[15] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," IEEE International Conference on Software Maintenance (ICSM), pp.382-391, 2011.

[16] X. Xia, L. Gong, T.D.B. Le, D. Lo, and J. Jiang,"Diversity maximization speedup for localizing faults in single-fault and multi-fault programs",

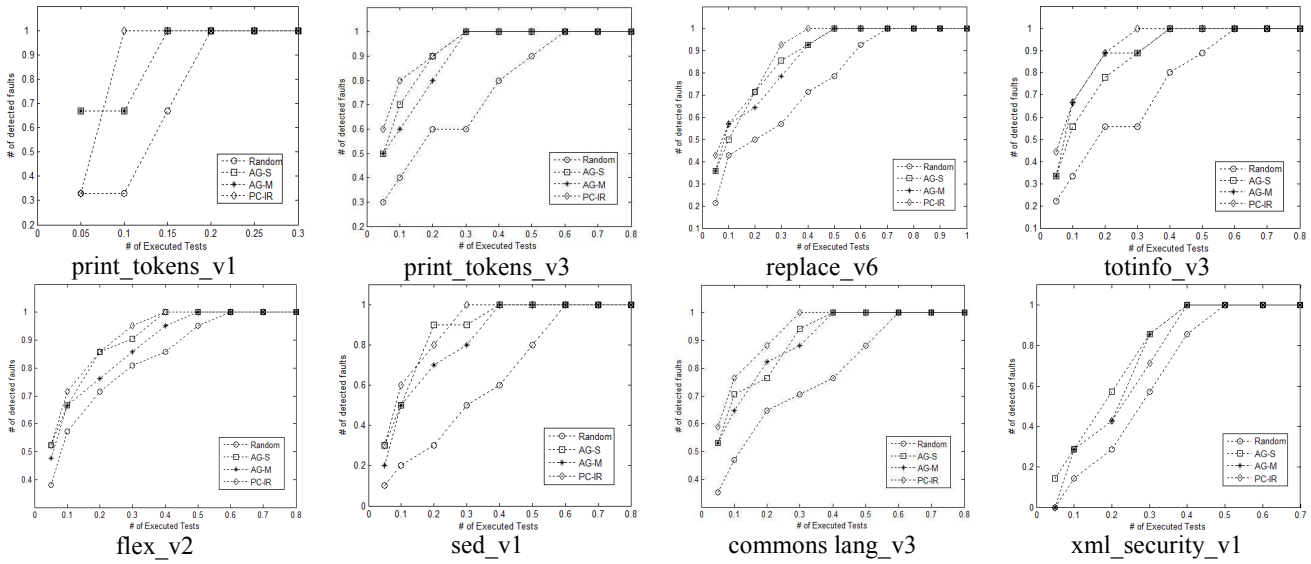

Figure 3. The number of faults detected by executed test cases for all TCP approaches

Automated Software Engineering, vol.23, no.1, pp.43-75, 2014.

[17] R. Wang, S. Jiang, and D. Chen, "Similarity-based regression test case prioritization." International Conference on Software Engineering and Knowledge Engineering, pp.358-363, 2015.

[18] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," IEEE International Symposium on Software Reliability, pp.81-90, 2007

[19] R.K. Saha, L. Zhang, S. Khurshid, and D.E. Perry, "An information retrieval approach for regression test prioritization based on program changes," IEEE/ACM International Conference on Software Engineering, pp.268-279, 2015.

[20] 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.

[21] Y.C. Huang, K.L. Peng, and C.Y. Huang, "A history-based costcognizant test case prioritization technique in regression testing," Journal of Systems & Software, vol.85, no.3, pp.626-637, 2012.

[22] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," 30th Annual International Computer Software and Applications Conference, pp.411-420, 2006.

[23] M. Yoon, E. Lee, M. Song, and B. Choi, "A test case prioritization through correlation of requirement and risk," Journal of Software Engineering & Applications, vol.5, no.10, pp.823-836, 2012.

[24] C. Hettiarachchi, H. Do, and B. Choi, "Risk-based test case prioritization using a fuzzy expert system," Information & Software Technology, vol.69, pp.1-15, 2016.

[25] Y. Ledru, A. Petrenko, and S. Boroday, "Using string distances for test case prioritization", IEEE/ACM International Conference on Automated Software Engineering, pp.510- 514, 2009.

[26] L. Zhang, S.S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming", 8th International Symposium on Software Testing and Analysis(ISSTA), vol.49, pp.665-669, 2009.

[27] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for Android apps", ACM International Symposium on Software Testing and Analysis, pp.425-436, 2016.

[28] 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, 2015.

[29] M.E. Delamaro and J.C. Maldanado, "Proteum- a tool for the assessment of test adequacy for C programs", The Conference on Perform ability in Computing System, 1996

[30] Y.S. Ma, J. Offutt, and Y.R. Kwon, "MuJava: an automated class mutation system: Research Articles", Software Testing Verification & Reliability, vol.15, no.2, pp. 97-133, 2005.

[31] X. Zhang, X. Xie, and T.Y. Chen, "Test case prioritization using adaptive random sequence with category-partition-based distance", IEEE International Conference on Software Quality, Reliability and Security, pp.374-385, 2016.

[32] A. Bosu, J.C. Carver, M. Hafiz M, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: an empirical study", ACM Sigsoft International Symposium on Foundations of Software Engineering, pp.257-268, 2014.

[33] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques", Information & Software Technology, vol.50, no. 6, pp. 534 – 546, 2008.

[34] CodeViz. http://freecode.com/projects/codeviz/

[35] WALA.http://wala.sourceforge.net/wiki/index.php/Main_Page/

[36] EMMA. http://emma.sourceforge.net/