Coverage-Based Clustering and Scheduling Approach for Test Case Prioritization

Wenhai FU(a), Student Member, Huiqun YU(b), Guisheng FAN(c), and Xiang JI(d), Nonmembers

SUMMARY 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 novel test case prioritization approach that combines the clustering algorithm and the scheduling algorithm for improving the effectiveness of regression testing. By using the clustering algorithm, test cases with same or similar properties are merged into a cluster, and the scheduling algorithm helps allocate an execution priority for each test case by incorporating fault detection rates with the waiting time of test cases in candidate set. We have conducted several experiments on 12 C programs to validate the effectiveness of our proposed approach. 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.

key words: regression testing, test case prioritization, clustering algorithm, scheduling algorithm

1. Introduction

As a kind of effective means to guarantee the quality of the software products, regression testing is used to convince that the validity of the modified part of the program, and the modified parts do not have any negative influence on other modules. However, rerunning all test cases in regression testing process may be time-consuming because of lots of test case executions. 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 trying to detect faults faster, and terminating 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 this paper, we propose a novel Clustering-Scheduling TCP approach to regression testing. This approach incorporates a clustering algorithm with a scheduling algorithm that utilizes historical execution information from previous testing versions. Based on the point that test cases with same or similar priorities have same or similar fault detection ability, we first cluster test cases that have same or similar properties into the same group according to code coverage information in historical executions. And then, following the point that test case which has a strong ability of detecting faults in previous versions will also perform better in a new testing version, we sort test cases in each cluster in descending order of their fault detection rates, and then initially rank test cases across entire clusters based on the rate of top test case in each cluster. The scheduling algorithm is used to determine the execution priorities of test cases by using two main functions: a dynamic priority adjustment strategy and a timeout response mechanism. Considering that code changes may affect the sensitivity of fault detection of test cases, the dynamic priority adjustment strategy exploits the results of the test cases that have been executed to estimate the fault detection rates of the remaining test cases. To detect all types of faults as soon as possible, the timeout response mechanism is established. A candidate test set is conducted to store test cases to be executed, and each test case in candidate test set should be given a waiting time. Once waiting time exceeds the maximum time limit, it should be the first to be executed regardless of the value of its fault detection rate. This timeout response mechanism can effectively reduce the risk of detecting a single type of fault.

To investigate the effectiveness of our approach, we have conducted empirical studies on 12 C programs, including 7 small-sized Siemens programs and 5 medium-sized programs. We compare our proposed approach with four other prioritization approaches which are popularly used to benchmark the performance of other prioritization approaches. Experimental results show that our proposed approach...
approach can effectively improve the effectiveness of regression testing.

2. Related Work

Researchers have proposed various metrics and techniques to improve the effectiveness of regression testing in recent years. The coverage-based TCP technique is one of the most widely studied techniques [6], [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. They are also popular as benchmarks to measure other techniques. In addition to these two strategies, researchers have also investigated other generic strategies. Li et al. [9] provide 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). 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.

Except for the above generic strategies using code coverage information, researchers have proposed various techniques based on types of code information, such as slices, code changes, or code modification information and fault proneness of code. Jeffrey and Gupta [15] present an algorithm that prioritizes test cases which combined statements coverage and relevant slices of the outputs of test cases. Khalilian et al. [16] prioritize test cases according to the available historical performance data which obtained from the previous test sessions. Huang et al. [17] introduce a technique that uses historical execution data to prioritize test cases in a regression testing process. Sherriff et al. [18] utilize change records to gather change impact information and prioritize test cases accordingly. Ripon et al. [19] introduce a REPIR approach by reducing the 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. Wang et al. [20] propose a global similarity-based regression test case prioritization approach based on the distance between pair-wise test cases.

There are also some approaches that do not utilize source code information. Kim et al. [12] propose a prioritization method based on test execution history to assign selection probabilities to test cases. Zhang et al. [21] propose a technique based on changing priorities of testing requirements and test case costs to prioritize test cases. Krishnamoorthy et al. [22] present a system-level test case prioritization using the information obtained from the requirements specification, such as requirements completeness and implementation complexity. Arafeen et al. [23] propose a document-clustering approach that incorporates requirements information into traditional code analysis information. Yoon et al. [24] identify more important test cases that are likely to detected effects associated with the system’s risks by using requirements risks. Hettiarachchi et al. [25], [26] propose a new requirements risk-based test case prioritization approach by considering the direct relationship between requirements risks and the test case.

As shown above, there are numerous TCP techniques using various types of information. In this paper, we utilize two types of information, code coverage and fault detection rates of test cases which are both helpful to improve the effectiveness of TCP techniques, to prioritize test cases. The code coverage information is one of the most widely used information when implementing test case prioritization techniques [6], [8]–[10], [14]. In our work, first, merging test cases into clusters is based on the similarities of their code coverage; second, we also use code coverage information to help prioritization when giving initial order of test cases in clusters and selecting test cases for execution. The fault detection information of test cases is an important kind of information which can directly reflect the ability of revealing faults of a test case, and the use of it can improve the effectiveness of TCP techniques [12], [17]. In our work, we use both the historical fault detection information of a test case and its estimated fault detection rate in the current version to help sort test cases.

Furthermore, works which are more relevant to our proposed approach generally take into account the correlation information between test cases which is helpful to improve the TCP technique. For example, test cases which have similar code structures or execution paths could detect similar faults [11], [27]. Experimental analysis in [28] has been also proved this point. Under the condition of limited resources, testers just need to detect a certain number of test cases in each cluster, and they still could have a better chance to detect more faults than otherwise. In [11], Leon and Podgurski propose a prioritization technique utilizing sampling methods that select test cases from clusters, which are formed based on distributions of test execution profiles. However, their technique simply selects test cases from clusters randomly for prioritization. Yoo et al. [27] investigate the use of expert knowledge to guide the ranking of test cases in clusters. The primary goal of their approach is to reduce human effort for pair-wise comparisons by using clustering. Carlson et al. [28] 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. However, they just sorted test cases in clusters by a traditional sorting method, and their approach could not adjust the priorities of the rest test cases dynamically using the latest execution information in this test phase.

3. Test Case Prioritization Approach

We now introduce our prioritization approach in this section. Our TCP approach contains two main steps. Firstly, we merge test cases that have the same or similar properties into the same cluster by comparing their historical code
coverage. Next, we gradually complete the prioritization process using a scheduling algorithm. The following subsections describe each of these steps in detail.

3.1 Clustering Approach

The test case clustering aims to merging the test cases with same or similar properties into the same cluster. We use an agglomerative hierarchical clustering method for clustering [29], which is based on the pair-wise distances between test cases in terms of code coverage similarity. Having known the code coverage (at statement-level granularity) of the entire test suite, it is not difficult to expand the corresponding vector of a test case. Given a program $P$, which consists of a number of executable statements $P = \{p_1, p_2, \ldots, p_m\}$, and a test suite $T = \{t_1, t_2, \ldots, t_n\}$ contains $n$ test cases. The execution trace of a test case $t_i$ $(1 \leq i \leq n)$ can be represented as a binary coverage vector, $C(t_i) = (s_1, s_2, \ldots, s_m)$, $m$ is the number of executable statements of the program $P$.

$$
s_k = \begin{cases} 
1, & \text{if the } k\text{-th executable statement covered by } t_i \\
0, & \text{otherwise} 
\end{cases}, \quad 1 \leq k \leq m
$$

If a statement is covered by a test case, we assign 1 to this statement in the coverage vector of this test case; otherwise, 0 is assigned. The vector $C(t_i)$ $(1 \leq i \leq n)$ represents the statement coverage of test case $t_i$. Given any two test cases $t_1$ and $t_2$, the distance between $t_1$ and $t_2$ is defined as the Euclidean distance between the coverage vectors $C(t_1)$ and $C(t_2)$.

$$
\text{Distance}(t_1, t_2) = \text{Euclidean}(C(t_1), C(t_2))
$$

By the property of the clustering algorithm, two test cases with the minimum distance are first combined into a cluster. Then the clusters are considered as a new element, the distances between two clusters are recomputed and produce new clusters. Actually, a test case would be merged with another test case, or with a cluster, or two clusters may be combined, based on the average distance between the two merging units. As a result, the clustering algorithm generates $k$ clusters, and each cluster contains from 1 to the total number of test cases.

3.2 Adjusting Fault Detection Rates of Test Cases

Test case prioritization process consists of two parts, i.e. the rank within each cluster and the rank between the clusters. The rank of test cases in each cluster can be determined at the beginning of the test case prioritization process and no longer changes. However, the final prioritized list for all test cases is generated gradually by selecting test cases from each cluster. Test cases detected faults in previous versions could have high fault detection abilities when they are reused to test the current version of the program. Hence, running test cases that have high fault detection ability earlier may improve the regression testing. Based on this point, we order the test cases in each cluster according to the historical fault detection ability in previous test versions. Furthermore, this ability of each test case would be also utilized as the criteria for test case prioritization.

We use fault detection rate to evaluate the fault detection ability of a test case. Test case that has a higher fault detection rate will be given a higher execution priority. Note that faults in the program are considered to have the same severity. The fault detection rate $r(t_i)$ of test case $t_i$ $(1 \leq i \leq n)$ can be defined as follows.

$$
r(t_i) = \frac{n(t_i)}{m}
$$

where $n(t_i)$ is the number of faults that are detected by test case $t_i$, $m$ is the total number of faults in the program. The bigger the value of $r(t_i)$, the higher the priority of $t_i$. We reorder test cases in each cluster simply by sorting them in descending order of their historical fault detection rates. The initial rank across entire clusters is also based on the fault detection rate of the top test case in each cluster. Note that, if there are different test cases whose $r(t_i)$ are the same, we will sort these test cases with additional greedy algorithm based on statement coverage.

Different from previous test case prioritization approaches which sort test cases just using the historical information, our approach gradually adjust the priorities of test cases based on the historical information and the feedback results of the test cases that are just executed. In the current version, test case which has been executed, its fault detection results can be used as the basis for adjusting the fault detection ability of the remaining test cases in the same cluster. Therefore, we first need to evaluate the fault detection rate of the latest executed test case. However, as the number and position of the faults in the program are both unknown, how to determine the fault detection rate of the test case that has just been executed is the key to our approach. We conjecture that all changed statement in the current program should be suspected of being faulty. First, we search the program differences between two versions at the line level by applying UNIX `diff` recursively while ignoring spaces and blank lines. Assume that each different line contains a fault, so the total number of faults in the current testing version can be quantified as the number of different lines. We denote the total number of faults as $dlln$. For a test case, its code coverage can be obtained easily, and we believe that it detects a fault as long as it has covered a different line in the program. So the number of faults that test case $t$ can be detected is quantified as the number of the different lines which it covers, and it denotes as $dlln(t)$. The estimated fault detection rate of test case $t$ in current testing versions can be modified as follows.

$$
r'(t) \approx \frac{dlln(t)}{dlln}
$$

Since test cases in the same cluster have the same or
similar properties, we conjecture that the fault detection rates of test cases in the same cluster also have the same or similar change trend. Based on this point, we calculate the relative deviation of the estimated fault detection rate with respect to the historical fault detection rate, and use this result as a basis for adjusting the fault detection rate of the next test case in the same cluster. Assuming that the \( j \)-th test case in the \( i \)-th cluster has been executed, \( r(t_i) \) and \( r'(t_i) \) represent the historical and estimated fault detection rates respectively, \( r(t_{i+1}) \) and \( r_p(t_{i+1}) \) represent the historical predictive fault detection rate of the \(( j + 1)\)-th test case in the \( i\)-th cluster. Based on the assumption that test cases in the same cluster have the same or similar change trend on the fault detection rates, we have the following formula.

\[
\frac{r'(t_i) - r(t_i)}{r(t_i)} = \frac{r_p(t_{i+1}) - r(t_{i+1})}{r(t_{i+1})}
\]

where \( r(t_i) \neq 0 \) and \( r(t_{i+1}) \neq 0 \). Therefore, the predictive fault detection rate of the next test case in cluster \( i \) can be adjusted as follows.

\[
r_p(t_{i+1}) = \frac{r(t_{i+1}) \times r'(t_i)}{r(t_i)}
\]

Note that Formula (6) is still tenable when \( r(t_{i+1}) = 0 \) which indicates that test case \( t_{i+1} \) can’t detect any faults in previous testing versions. That is to say, when \( r(t_i) \neq 0 \) and \( r(t_{i+1}) = 0 \), \( r_p(t_{i+1}) \) takes a value of 0 regardless of the change of fault detection rate of \( t_i \) in the current version.

Now we discuss how to adjust \( r_p(t_{i+1}) \) if \( r(t_i) \) is 0. As we know, test cases in the same cluster is sorted in descending order of their historical fault detection rates, so for test cases \( t_i \) and \( t_{i+1} \), \( r(t_i) \) is always no less than \( r(t_{i+1}) \). If \( r(t_i) \) takes a value of 0, \( r_p(t_{i+1}) \) is also 0. Taking into account the criteria for selecting test cases in the next phase (details can be found in Sect. 3.4), under such a circumstance, we stipulate that the change of \( r(t_i) \) will not impact the fault detection rate of \( t_{i+1} \).

In summary, the predictive fault detection rate of the next test case \( t_{i+1} \) can be calculated as follows:

\[
r_p(t_{i+1}) = \begin{cases} 
\frac{r(t_{i+1}) \times r'(t_i)}{r(t_i)}, & r(t_i) \neq 0 \\
0, & otherwise 
\end{cases}
\]

From Formula (7), we can see that the maximum value of \( r_p(t_{i+1}) \) is 1, and the minimum value is 0.

We use a simple example to illustrate how we calculate the predictive fault detection rate for the next test case after executing the last test case in the same cluster. Assuming that there are two adjacent test cases \( t_i \) and \( t_{i+1} \) in the same cluster \( C_i \), the historical fault detection rates of them are 0.5 and 0.4, respectively. Executing test case \( t_i \) on the current program version, the estimated fault detection rate is 0.3. According to Formula (7), the predictive fault detection rate of \( t_{i+1} \) can be calculated as \( r_p(t_{i+1}) = (0.3 \times 0.4)/0.5 = 0.24 \), the ability of \( t_{i+1} \) detecting faults becomes weak on the current program version.

### 3.3 A Timeout Response Mechanism

To perform test case prioritization, firstly, we create a candidate test set which is used to store test cases with the highest fault detection rates in each cluster. The scale of this candidate test set is set to the total number of clusters. All test cases for prioritization should be selected from the candidate set. The initial candidate set is composed of the top test case in each cluster. Since the criteria for test case prioritization in our approach is based on the abilities of fault detection of test cases, the process of test case prioritization is as follows:

1. Select and execute the test case which has the maximum fault detection rate from the candidate test set, and if there are a number of test cases which have the same fault detection rate in the candidate set, the test case which cover most of the statements is the optimal selection; 2) Once the test case is selected to be executed, it should be removed from the candidate set, and at the same time, the next test case in the same cluster will be added into the candidate set and its predictive fault detection rate should be adjusted according to Formula (7); 3) Repeat the above two steps until all the test cases are executed.

If test cases are prioritized using the above method, the prioritization may be faced with such a situation: the test case which is selected from the candidate set each time is always from the same cluster. That is, test cases in one cluster always have higher priorities than test cases in other clusters. Until test cases with higher priorities in this cluster have been prioritized, test cases in other clusters could be considered to be prioritized. Since test cases in the same cluster have the same or similar code coverage, there is a great possibility that these test cases would detect same faults. Therefore, this situation will make the same faults be revealed repeatedly over a period of time. Obviously, the effectiveness of our approach will be greatly reduced. Therefore, we introduce a timeout response mechanism to avoid this risk.

For each candidate test case, we give a waiting time when it enters the candidate set. This waiting time indicates the time that a test case has waited for being executed in the candidate set. Meanwhile, we set a maximum waiting time \( \text{wt}_{\text{max}} \) for the test cases in the candidate set. Once the waiting time of a test case exceeds the maximum time limit, it should be the first to be executed regardless of its fault detection rate. This timeout response mechanism can effectively reduce the risk of detecting a single type of fault.

We assume that the execution time of each test case is the same, and the time of an execution can be regarded as a time unit. Thus, the maximum waiting time \( \text{wt}_{\text{max}} \) can be set to a number of time units. Using \( W_c \) to represent the candidate test set, a test case \( t_i \) is selected from \( W_c \) for execution if:

1. its waiting time has exceeded \( \text{wt}_{\text{max}} \), \( \text{wt}(t_i) > \text{wt}_{\text{max}} \);
2. or \( (2) \) its fault detection rate is highest and there is no test case in \( W_c \) whose waiting time exceeds \( \text{wt}_{\text{max}} \).

For the condition (1), if there are multiple test cases
whose waiting time has exceeded \( w_{t_{\text{max}}} \), we should also consider the following two cases: a) when their waiting time is all the same, test case with the highest fault detection rate should take the first place to be selected, b) otherwise, test case with the longest waiting time should be selected.

For the condition (2), if there are multiple test cases which have the same fault detection rates, we will select one the test case which covers most of the statements.

Once a test case is selected to be executed, we need to perform the following operations: 1) Firstly, remove the selected test case from the candidate set; 2) Add the next test case in the same cluster with the selected test case into the candidate set with an initial waiting time, and calculate its predictive fault detection rate according to Formula (7); 3) Finally, update the waiting time for the rest test cases in candidate set.

### 3.4 Summary of Our TCP Approach

Algorithm 1 is the main C-S (Clustering-Scheduling) algorithm, and it contains two main steps. Firstly, we use a clustering algorithm to merge test cases that have the same or similar coverage properties into clusters (line 1); next, we gradually complete the TCP process by using a scheduling algorithm (lines 7 to 13). The scheduling algorithm contains two main functions that determine the execution priorities of test cases: 1) A dynamic priority adjustment strategy that exploits the results of the tests that have been executed to estimate the fault detection rate of the remaining test cases; 2) A timeout response mechanism to make sure that more types of faults can be detected as soon as possible. Finally, C-S returns the resultant prioritized sequence (line 14). Note that, since our approach uses the testing results information produced by the current testing version to guide the subsequent testing, the process of test case prioritization is also the testing process. Therefore, when the algorithm terminates, the testing process of current version has been completed.

Our approach combines the advantages of total greedy and additional greedy algorithm because it takes into account not only the number of faults but also the diversity of faults that a test case can detect. By evaluating the fault detection rates of test cases, we give a high priority to the test case with high fault detection rate. Moreover, we introduce an information feedback mechanism, which combines the historical data with the execution information of test case which has been executed in current testing version, to evaluate more accurate fault detection rates for the rest similar test cases in the same cluster. By introducing the timeout response mechanism, our approach ensures that different types of faults can be detected as soon as possible.

We use a simple example to illustrate our proposed approach. In Fig. 1, test suite \( T \) consists of \( n \) test cases, and has been merged into four clusters, \( W_c \) and \( W_p \) represent the candidate test set and executed test set respectively. Initial order has been done according to the historical fault detection rates. The scale of candidate set is 4, and the maximum waiting time \( (w_{t_{\text{max}}}) \) is set to be 4 time units.

\[
\text{Wc} = \{t_1, t_2, t_3, t_4\} \quad \text{Wp} = \{t_5, t_6\}
\]

The fault detection rate of \( t_1 \) is 0 (i = 1, 2, 3, 4). According to the priorities of test cases, test case \( t_1 \) is executed first, and the next test case \( t_3 \) after \( t_1 \) is added into \( W_c \). The fault detection rate of \( t_3 \) is adjusted according to formula (7) with the result of \( t_1 \). At the same time, we update the waiting time for \( t_2, t_1, t_4, t_5, w(t_2) = w(t_3) = w(t_4) = 0 \). Repeat above steps, test case \( t_2 \) which satisfies the selection condition \( (t_2 \) has the highest fault detection rate and the waiting time of test cases in \( W_c \) is all less than 4) is executed and \( t_6 \) is added into \( W_c \). Likewise, we adjust the fault detection rate of \( t_6 \) with the result of \( t_2 \), and update the waiting time of test cases in \( W_c \), \( w(t_6) = w(t_4) = 2 \), \( w(t_5) = 1 \), \( w(t_6) = 0 \). Repeat above steps until the fifth iteration, \( w(t_4) \) has been over \( w_{t_{\text{max}}} \). In this iteration, the waiting time should be considered as the first factor. So in the next iteration, although the fault detection rate of \( t_4 \) is not the highest, it is selected to be executed in this iteration. The algorithm terminates until all test cases

**Algorithm 1 : C-S Algorithm**

**Inputs:** \( T \) a set of test cases \( \{t_1, t_2, \ldots, t_n\} \)

\( R \) : the historical fault detection rates of test cases in \( T \), \( r(t_1), r(t_2), \ldots, r(t_n) > k \) : the number of clusters

\( w_{t_{\text{max}}} \) : the maximum waiting time

**Output:** \( P \) : a sequence of prioritized test cases \( \langle p_1, p_2, \ldots, p_n \rangle \)

1. Using the clustering algorithm to produce \( k \) clusters \( \langle C_1, C_2, \ldots, C_k \rangle \)
2. for \( i = 1, 2, \ldots, k \)
3. Sort(C) // Initialization ranking based on the historical fault detection rates in descending order

4. end
5. Initialization(\( W_c \))
6. \( P = \{\} \)
7. while \( |W_c| > 0 \) do
8. Schedule test case \( t_i (1 \leq i \leq k, 1 \leq j < n) \) from \( W_c \) for execution if:
9. \( (1) \ w(t_i) > w_{t_{\text{max}}} \), or \( (2) \ w(t_i) \leq w_{t_{\text{max}}} \) and \( r(t_i) \) is the highest in \( W_c \)
10. \( W_c = W_c \setminus \{ t_i \} \) \{ \{ t_{j=1} \} \}
11. Update waiting time for the rest test cases in \( W_c \):
12. \( w(t) \leftarrow w(t)+1, w(t_{j=1})=0 \)
13. Adjust \( r(t_i) \)
14. \( P \leftarrow P + \{t_i\} \)
15. end
16. return \( P \)

Fig. 1 A simple example for test case prioritization
have been executed.

3.5 Discussion

Maximum Waiting Time. We further analyze the effect of the maximum waiting time ($wt_{max}$) in this timeout response mechanism. As we know, $wt_{max}$ is used to help detect more types of faults as soon as possible. If the value of $wt_{max}$ is too large, the impact of the waiting time on limiting test case prioritization will be significantly weakened. It means that our algorithm tends to prioritize test cases only based on the fault detection rates of test cases. This makes our approach be similar to a total greedy algorithm based on the fault history information. On the contrary, if the value of $wt_{max}$ is too small, there will be more and more test cases in candidate test set whose waiting time exceeds the maximum waiting time, and all these test cases should be prioritized. According to the prioritizing condition of our approach, these test cases should be prioritized based on the length of waiting time. As test cases from different hierarchies are continually added into the candidate test set, their waiting time will be also different. Waiting time of test cases from a hierarchy must be longer than that of test cases from the next hierarchy, until test cases in the upper hierarchy have all prioritized, test cases from the next hierarchy are considered to be executed. This situation is similar to prioritize test cases layer-by-layer in accordance with their priorities in their clustering hierarchy. We will discuss the effect of the maximum waiting time on the effectiveness of our approach by experiments in Sect. 4.

4. Empirical Study

4.1 Research Questions

To evaluate the effectiveness of our TCP approach on fault detection, we design and implement a set of empirical studies to answer the following research questions.

RQ1: Is our proposed test case prioritization technique effective in improving the rate of fault detection?

RQ2: How does the maximum waiting time impact the effectiveness of our approach?

4.2 Subject Programs

Our experiments are conducted on 12 subject programs, including 7 small-sized Siemens programs and 5 medium-sized programs. The Siemens suite is composed of seven programs. Each program has several faulty versions, and each version contains exactly one fault which may span through multiple statements and functions. Every program has exactly or more than 1,000 tests. Space functions as an interpreter for an array definition language (ADL). Flex is used to recognize lexical patterns in the text; sed is a stream editor which parses the input text, applies some operations, and then outputs the modified text; grep is mainly used to capture the pattern matching line; gzip is a file compression program. All these programs can be downloaded from http://sir.unl.edu. We consider a number of consecutive versions for each program. The faults of each version are seeded by hand, and each version contains numbers of faults from 0 to 31. The method of generating each faulty version is similar to the method used by Elbaum et al. [31]. As we know, Siemens and space programs initially have a number of fault versions containing 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. However, due to program changes and faults merge, some test cases cannot be executed successfully and they should be excluded from the test pool. We run each multi-fault version on the original test pool, and construct its unique test suite. Since the number of test cases of space program is too large, we randomly select 2000 test cases from its test pool for each faulty version. A test case is selected as long as the test case can increase the coverage of the suite. The process terminates until all executable statements have been covered. This procedure is also used in [14], [32]. For each UNIX program, we generate the new faulty versions by fixing faults or merging other faults in the original versions provided by SIR. Table 1 shows the details of the subject programs used in our experiments.

4.3 Techniques for Comparison

To evaluate the performance of our proposed approach, random prioritization, total greedy prioritization algorithm, additional greedy prioritization algorithm, and a clustering approach using code coverage information in [28] are also used for comparison in our experiments. As mentioned in Sect. 2, the first three approaches are popularly used to benchmark other TCP approaches, and among them, additional greedy algorithm is widely considered as the most effective TCP approach to quickly expose faults from a program in terms of fault detection capacity [8]. TCP algorithm in [28] uses a clustering algorithm with three kinds of historical data. Table 2 summarizes these approaches. All total greedy, additional greedy, Tcov-clst algorithm and our proposed algorithm utilize code (statement) coverage information. Random algorithm orders test cases just one by

| Program | No. of Faulty Versions | Lines of Code | No. of Test Cases |
|---------|------------------------|---------------|-------------------|
| print tokens | 10 | 720 – 726 | 4071 |
| print tokens2 | 8 | 557 – 563 | 4115 |
| replace | 15 | 563 – 566 | 5562 |
| schedule | 10 | 360 – 369 | 2650 |
| schedule2 | 10 | 329 – 335 | 2710 |
| csc | 8 | 162 – 176 | 1608 |
| bfinfo | 9 | 540 – 560 | 1052 |
| space | 17 | 9420 – 9562 | 13585 |
| flex | 15 | 10054 – 12407 | 567 |
| sed | 15 | 4711 – 9289 | 570 |
| grep | 18 | 8053 – 9089 | 809 |
| gzip | 12 | 4731 – 5159 | 217 |


one randomly. Since random approach is based on random selection, and it could generate different test case orders for regression testing, we repeat it 50 times for each regression testing to obtain averages that can portray typical performance.

4.4 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.

\[
APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_n}{nm} + \frac{1}{2n}
\]  

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.

4.5 Experiment Process

In order to perform prioritization, we need two data sets, i.e. code coverage information and fault history records. We collected the code coverage data for test cases executed over its test pool by some instrumentation tool such as “gcov” command of GNU C compiler to. To collect fault history information, we record the fault detection rate of each executed test case from previous versions. Since the codes in each test version have been changed differently, the codes that were covered by even same test cases in different test versions are not same; neither does the fault detection information. In theory, the test version whose code change is similar to the current test version, its testing results (code coverage and historical fault detection information) can be used as the historical information in current testing version. Once we collected all the required data, we merge test cases into clusters by using an agglomerative hierarchical clustering method that we explained in Sect. 3.1. To prevent excessive merge, we set the termination condition for clustering

Table 2 Prioritization approaches used in our empirical study

| Algorithm                | Description                                      |
|--------------------------|--------------------------------------------------|
| Random                   | Randomly orders the test cases one by one        |
| Total Greedy (TG)        | Sort test cases in descending order of the total number of program constructs covered |
| Additional Greedy(AG)    | Sort test cases in descending order of the program elements that not yet covered by the selected test cases |
| Code coverage using clustering (Tcov-clust) | First cluster test cases and then using clustered test cases, prioritize test cases based on code coverage |
| Clustering-Scheduling (C-S) | Our proposed technique |

based on the scale of test cases of each program and the intermediate experimental data in our previous work [30]. Since the Siemens programs and space programs have a large number of test cases, we define that when the current number of clusters is approximately 1% of the initial number of clusters, the merging is terminated and the clustering is completed. Likewise, considering the scales of test cases of the other four programs, we create ten clusters for these programs.

We implement the five different prioritization techniques described in Sect. 4.3 to prioritize test cases for each program. When answering RQ1, we set 0.75k as the maximum waiting time for printtokens and space, 2k time units as the maximum waiting time for schedule, tcus, and rotinfo, and k time units for other programs (k is the number of clusters that test cases are merged in each program). Then, the APFD values are computed from the reordered test cases, and the values are analyzed to identify whether the proposed prioritization approach can improve the rate of fault detection.

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.

4.6 Experimental Results Analysis

(1) **Answer RQ1**

Our research question considers whether our approach (C-S) is effective, we compare it with some of the best test case prioritization techniques in terms of APFD. Table 3 shows the APFD values of the experimental results of each approach for all subject programs, respectively. In Table 3, 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 C-S over previous approaches. For example, the mean APFD value of random technique for printtokens is 73.71%, while the APFD value produced by C-S is 89.07%. The improvement APFD value of C-S over Random technique is 15.36%.

Let’s analyze the APFD values of the prioritization approaches in the experiment in detail. As seen from Table 3, the average APFD value of C-S is greater than those of other techniques on all programs, except additional greedy technique on replace, schedule, and grep. It means that C-S outperform other approaches across majority of program versions which does not consider test case costs and fault severities. Comparing C-S with total greedy algorithm (TG), C-S has a significant improvement on mean APFD (improvement of more than 20% on each program). C-S also has a significant improvement in comparison with random technique, especially on the small-sized programs (improvement of more than 15%). When compared with other clustering approaches based on coverage information, our proposed C-S approach also performs outstandingly (improvement of more than 5% on four programs, more than 3% on
nine programs). Even compared with the most effective additional greedy prioritization approach, our approach also performs better on 9 of 12 programs, and improvements are significant on 4 programs. This proves that our proposed approach is effective in improving the rate of fault detection.

Further, to show our results visually, boxplots for each technique for programs with their APFD values are shown in Fig. 2. The figure contains 12 subfigures, and each subfigure shows the distribution of APFD values of all five prioritization approaches for a program. The horizontal axis corresponds to TCP approaches, and the vertical axis corresponds to APFD values. From Fig. 2, we observe that C-S approach perform outstandingly. We observe that our approach is significantly more effective than random technique, the total greedy technique, and the clustering-based technique. Moreover, except for the case on schedule, replace, and grep, we find that the median APFD of C-S is visually more effective than the additional greedy technique. Examining the boxplots for each approach in Fig. 2, the trends observed from Table 3 (average values) are consistent with the results shown from the boxplots. In particular, for all five approaches, the differences between the best and worst APFD values are noticeable.

To compare our approach with the two more

| Program | Approach | Mean | Improvement | Program | Approach | Mean | Improvement | Program | Approach | Mean | Improvement |
|---------|----------|------|-------------|---------|----------|------|-------------|---------|----------|------|-------------|
| print_tokens | Radom | 73.71 | 15.36 | Radom | 70.34 | 15.76 | Radom | 71.52 | 13.61 |
|         | TG | 64.88 | 24.19 | TG | 62.85 | 23.25 | TG | 62.44 | 22.69 |
|         | AG | 87.80 | 1.27 | AG | 82.91 | 3.19 | AG | 87.07 | -0.94 |
|         | Cov-clst | 85.94 | -3.23 | Cov-clst | 83.06 | -3.04 | Cov-clst | 82.18 | 2.95 |
|         | C-S | 89.07 | -- | C-S | 86.1 | -- | C-S | 85.3 | -- |
| schedule | Radom | 66.37 | 16.25 | Radom | 68.5 | 17.94 | Radom | 53.2 | 21.8 |
|         | TG | 61.94 | 20.68 | TG | 59.47 | 26.97 | TG | 46.58 | 28.48 |
|         | AG | 84.22 | -1.6 | AG | 81.08 | 5.36 | AG | 75.2 | 1.86 |
|         | Cov-clst | 77.46 | 5.16 | Cov-clst | 78.17 | 8.27 | Cov-clst | 73.52 | 1.54 |
|         | C-S | 82.62 | -- | C-S | 86.44 | -- | C-S | 75.06 | -- |
| totinfo | Radom | 71.96 | 13.77 | Radom | 73.45 | 15.52 | Radom | 74.68 | 14.72 |
|         | TG | 59.74 | 25.99 | TG | 65.19 | 23.78 | TG | 66.73 | 22.67 |
|         | AG | 80.35 | 5.38 | AG | 82.3 | 6.67 | AG | 87.73 | 1.67 |
|         | Cov-clst | 82.04 | 3.69 | Cov-clst | 78.4 | 10.37 | Cov-clst | 84.55 | 4.87 |
|         | C-S | 85.73 | -- | C-S | 88.97 | -- | C-S | 89.4 | -- |
| sed | Radom | 77.26 | 12.59 | Radom | 73.88 | 12.38 | Radom | 71.51 | 15.09 |
|         | TG | 67.74 | 22.61 | TG | 60.13 | 26.13 | TG | 58.28 | 28.32 |
|         | AG | 88.18 | 1.67 | AG | 87.53 | -1.27 | AG | 84.67 | 1.93 |
|         | Cov-clst | 85.93 | 3.92 | Cov-clst | 84.91 | 1.35 | Cov-clst | 80.45 | 6.15 |
|         | C-S | 89.85 | -- | C-S | 86.26 | -- | C-S | 86.6 | -- |

Fig. 2 APFD distributions for all subject programs
effective TCP approaches (additional greedy approach and Tcov-clst approach) more intuitively and in more detail, taking print_tokens and flex as examples, the bar charts of APFD values on each faulty version are given in Fig. 3. From Fig. 3, we observe that C-S 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 trying 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. Figure 4 shows the results of this relationships for all five TCP approaches on version 2 of tcas program (contains 12 faults), version 1 and version 3 of space (contain 23 and 29 faults respectively), version 5 of grep (contains 30 faults). From Fig. 4, we can observe that when executing the same percentage of test cases, our approach can always detect more faults. For example, when 20% test cases have been executed on version 2 of tcas program, more than 90% faults can be detected using our TCP approach. This result also proves that our approach can improve the rate of fault detection.

(2) Answer RQ2

The results of the prior study prove that our proposed approach can help improve the effectiveness of prioritization, and our approach outperforms other algorithms when given a specified value of the variable factor—the maximum
Fig. 5 APFD distributions when giving different wtmax values for all subject programs

waiting time. RQ2 mainly investigates the influences on the effectiveness of our proposed approach by the change of this factor.

In Sect. 3.5, we discuss the impact of this factor on the effectiveness of our approach. In order to verify the discussion, we design and conduct the experiments on the 12 programs by giving different maximum waiting time for them, and compare the final results with the pure fault-based total greedy algorithm (Tfb) and Tfb-clst algorithm in [28]. The use of clustering in test case prioritization is proved that can improve the effectiveness of prioritization in terms of increasing the rate of fault detection in [28], which is, Tfb-clst algorithm is more effective than fault-based total greedy algorithm.

In this experiment, we set 0.25k, 0.5k, 0.75k, k, 2k, 3k, 5k, 10k time units to the maximum waiting time (wtmax), k represents the number of clusters which test cases in each program have been merged. Figure 5 shows the results of 6 out of 12 programs when wtmax is given different values. (These 6 programs represent different types of benchmark programs: small-sized programs with a large or a relatively few number of test cases and medium-sized programs with a large or a relatively few number of test cases.) From Fig. 5, we can see that when wtmax takes a small value, the APFD values of our approach are close to the APFD values of Tfb-clst in [28]. On the contrary, when wtmax is too large (more than 5k time units), the results of our experiments in terms of APFD are similar to the values of fault-based total greedy algorithm. The results of our experiments are consistent with the aforementioned theoretical analysis in Sect. 3.5. We also observe that when wtmax is given an appropriate value, our approach will obtain a better result. For example, when wtmax is given a value of 0.75k time units on print_tokens program, our approach is more effective than the Tfb-clst algorithm and Tfb algorithm, even than the additional greedy algorithm. And when wtmax takes a value of k time units on flex program, the APFD value of our approach is the best. This conclusion has been verified in RQ1.

From the results of the experiments, we observe that the effectiveness of our approach C-S will be influenced by the maximum waiting time. Although the effectiveness of C-S is not stable by comparing with the additional greedy algorithm, it is more effective than random algorithm, total greedy algorithm, Tfb-clst algorithm in general. When wtmax is given the optimal value, our algorithm will be better than the additional greedy algorithm on most of the faulty program versions. The results also show that the effectiveness of C-S is better than that of the Tfb-clst algorithm when the value of wtmax is between the minimum value and the optimal value; similarly, when the value of wtmax is between the optimal value and the maximum value, C-S is more effective than the total greedy algorithm.

Furthermore, from the results of RQ2, we observe that the effectiveness of the algorithm in terms of APFD may obtain the best value when the value of wtmax is in the range of 0 to the half most number of test cases in the cluster. We speculate that this result is more related to the similarity of test cases in each cluster. Although we are not sure whether this conclusion is applicable to the clustering results obtained by different clustering methods or different number of clusters, it provides a research direction for us to find the optimal value of wtmax.

4.7 Time Overhead

Our proposed TCP approach contains two main steps:
Table 4  Average execution time of C-S algorithm and additional greedy algorithm (AG)

| Programs    | C-S (s)  | AG (s)  |
|-------------|----------|---------|
| print_tokens| 11.245   | 115.260 |
| print_tokens2| 11.507  | 112.726 |
| replace      | 14.818   | 220.290 |
| schedule     | 7.285    | 55.228  |
| schedule2    | 6.857    | 51.390  |
| cas          | 3.908    | 17.406  |
| scanf        | 2.845    | 7.136   |
| space        | 13.454   | 963.881 |
| lex          | 5.778    | 57.868  |
| sed          | 3.589    | 14.237  |
| grep         | 6.432    | 110.890 |
| egrep        | 3.074    | 6.610   |

(1) merging test cases that have the same or similar properties into the same cluster; (2) selecting test cases for execution from the candidate test set. The time cost of C-S approach also depends on these two procedures.

Since the cluster algorithm used in our approach has a high time complexity, it will take a long time for clustering test cases, especially for the programs with a large number of test cases. For example, clustering test cases for the small-sized program print_tokens which has approximately 4000 test cases, the mean time for clustering is 416.315s.

To reduce the time cost of clustering, we use the same test case clusters for test case prioritization on several subsequent faulty versions which have same test suite and similar code coverage.

The time cost of selecting test cases for execution from the candidate test set consists of the following two parts: 1) the time of selecting test cases from the candidate test set; 2) the time of executing test cases and collecting feedback of executed test cases. Since our approach selects the test case from the candidate test set each time, the time of selecting test case from candidate set is very small (the time of selecting a test case from a candidate test set which has 50 test cases is 0.0068ms). However, C-S approach is still not efficient when compared with other test case prioritization techniques. Because in our approach, we need to collect the coverage information of the new executed test cases for adjusting the fault detection rates of the rest test cases, and code coverage profiling can be costly. In our experiments, the execution time of our C-S approach is always more than the time of random algorithm, total greedy algorithm, and Tcov-clst algorithm, but still less than the time used by additional greedy algorithm. Table 4 shows the average time cost of C-S algorithm and additional greedy algorithm which is considered to be the most costly TCP algorithm on different programs. From Table 4, we can see that our approach is always more efficient than additional greedy algorithm.

Although our approach need to takes more time than random algorithm, total greedy algorithm, and Tcov-clst algorithm, there is no doubt that our approach can be most efficient in testing continuous program versions. When prioritizing test cases using code-coverage-based techniques, we generally need to obtain the historical testing coverage information before testing a new program version, and this will also be costly. Because collecting coverage information of the test cases has been completed with the test case prioritization in the current testing round, there is no additional time overhead when testing for the next program version in our C-S approach. If we only consider the time needed by prioritizing test cases, our approach is much more efficient than existing coverage-based techniques.

4.8 Threats to Validity

The main threats to internal validity are as follows. The number of clusters chosen in our experiments could affect the results of our study. The number of clusters in our experiments is determined based on the scale of the programs used in the experiments. In our work, experiments are conducted in 12 object programs in C, and thus, we are unable to definitively state that our approach will hold for other larger programs written in different languages in general. We used APFD to measure the effectiveness of the studied test case prioritization techniques. Using other metrics may provide different results. Again, this limitation can be addressed through additional studies with different numbers of clusters and different maximum waiting times.

Threats to external validity deals with the subject programs used in the experiments. In our work, experiments are conducted in 12 object programs in C, and thus, we are unable to definitively state that our approach will hold for other larger programs written in different languages in general. We used APFD to measure the effectiveness of the studied test case prioritization techniques. Using other metrics may provide different results. Again, this limitation can be addressed through additional studies with a wider population.

The main threat to construct validity comes with whether the experiments are measured in a correct way. In our empirical study, we compared C-S with existing code-coverage-based TCP techniques only considering their APFD values. The comparison results reported in Table 3 are valid in that the C-S techniques are more effective than the three previous techniques. However, in our experiments, we only measure the effectiveness of our proposed approach without considering other factors. In future work, we will apply a novel measurement for test case prioritization approach by considering more factors.

5. Conclusion

In this paper, we propose a new test case prioritization based on clustering and scheduling algorithms. By clustering, test cases which have the same or similar properties will be merged into the same clusters, which can help guide the fault
detection. By scheduling, assigning execution priorities for test cases based on predictive fault detection rates and waiting time in candidate set is conducive to detect all faults as soon as possible. Our experiments are conducted on 12 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 total greedy algorithm, additional greedy algorithm and a clustering approach using code coverage information. Further experiments also show that the uncertainty factor in our approach, the maximum waiting time, will affect the effectiveness of our approach.

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.

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 Trans. Softw. Eng. Methodol., 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 Trans. Softw. Eng., vol.33, no.2, pp.108–123, 2007.
[3] H. Mei, D. Hao, L. Zhang, L. Zhan, J. Zhou, and G. Rothermel, “A static approach to prioritizing JUnit test cases,” IEEE Trans. Softw. Eng., 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,” 21st 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 Trans. Softw. Eng., 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,” J. 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 Trans. 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 Trans. Softw. Eng., 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 Trans. Softw. Eng., 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 (ICSE), 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] D. Jeffrey and N. Gupta, “Test case prioritization using relevant slices,” 30th Annual International Computer Software and Applications Conference, pp.411–420, 2006.
[16] 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.
[17] Y.C. Huang, K.L. Peng, and C.Y. Huang, “A history-based cost-cognizant test case prioritization technique in regression testing,” J. Systems & Software, vol.85, no.3, pp.626–637, 2012.
[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.91–90, 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] 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.
[21] X. Zhang, C. Nie, B. Xu, and B. Qu, “Test case prioritization based on varying testing requirement priorities and test case costs,” International Conference on Quality Software, pp.15–24, 2007.
[22] R. Krishnamoorthi and S.A. Mary, “Factor oriented requirement coverage based system test case prioritization of new and regression test cases,” Information & Software Technology, vol.51, no.4, pp.790–808, 2009.
[23] M.J. Arafeen and H. Do, “Test case prioritization using requirements-based clustering,” 6th International Conference on Software Testing, Verification and Validation, pp.312–321, 2013.
[24] M. Yoon, E. Lee, M. Song, and B. Choi, “A test case prioritization through correlation of requirement and risk,” J. Software Engineering & Applications, vol.5, no.10, pp.823–836, 2012.
[25] C. Hettiarachchi, H. Do, and B. Choi, “Effective regression testing using requirements and risks,” 8th International Conference on Software Security and Reliability, pp.157–166, 2014.
[26] 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.
[27] S. Yoo, M. Harman, P. Tonella, and A. Susi, “Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge,” International Symposium on Software Testing & Analysis (ISTTA), pp.201–212, 2009.
[28] R. Carlson, H. Do, and A. Denton, “A clustering approach to improving test case prioritization: An industrial case study,” 27th IEEE International Conference on Software Maintenance (ICSM), pp.382–391, 2011.
[29] P. Tan, M. Steinbach, and V. Kumar, “Introduction to Data Mining,” Addison-Wesley Longman Publishing, 2005.
[30] W.H. Fu, H.Y. Yu, and G.S. Fan, “A test case selection approach to improving the effectiveness of fault localization,” J. East China University of Science and Technology (Natural Science Edition), vol.42, no.4, pp.557–562, 2016.
[31] S. Elbaum, A.G. Malishevsky, and G. Rothermel, “Test case prioritization: a family of empirical studies,” IEEE Trans. Softw. Eng., vol.28, no.2, pp.159–182, 2002.
[32] B. Jiang, W.K. Chan, and T.H. Tse, “PORA: Proportion-Oriented Randomized Algorithm for Test Case Prioritization,” IEEE International Conference on Software Quality, Reliability and Security, pp.131–140, 2015.
Wenhao Fu received her B.S. degree from East China University of Science and Technology (ECUST) in 2012 in computer science. She is a Ph.D. student in computer science at East China University of Science and Technology. Her current research interests include software engineering and software fault localization.

Huiqun Yu received his B.S. degree from Nanjing University in 1989, M.S. degree from East China University of Science and Technology (ECUST) in 1992, and Ph.D. degree from Shanghai Jiaotong University in 1995, all in computer science. He is currently a Professor of computer science with the Department of Computer Science and Engineering at ECUST. From 2001 to 2004, he was a Visiting Researcher in the School of Computer Science at Florida International University. His research interests include software engineering, high confidence computing systems, cloud computing and formal methods. He is a member of the ACM, a senior member of the IEEE, and a senior member of the China Computer Federation.

Guisheng Fan received his B.S. degree from Anhui University of Technology in 2003, M.S. degree from East China University of Science and Technology (ECUST) in 2006, and Ph.D. degree from East China University of Science and Technology in 2009, all in computer science. He is presently a research assistant of the Department of Computer Science and Engineering, East China University of Science and Technology. His research interests include formal methods for complex software system, service oriented computing, and techniques for analysis of software architecture.

Xiang Ji received his B.S. degree from East China University of Science and Technology (ECUST) in 2013. He is a Ph.D. student in computer science at East China University of Science and Technology. His research interests include software engineering, formal methods and Cyber Physical System.