History-Based Dynamic Test Case Prioritization for Requirement Properties in Regression Testing

Xiaolin Wang Hongwei Zeng
School of Computer Engineering and Science, Shanghai University
Shanghai, 200444, China
+86-021-66135750
{wangxiaolin, zenghongwei}@shu.edu.cn

ABSTRACT
Regression testing is an important but extremely costly and time-consuming process. Because of limited resources in practice, test case prioritization focuses on the improvement of testing efficiency. However, traditional test case prioritization techniques emphasize only one-time testing without considering huge historical data generated in regression testing. This paper proposes an approach to prioritizing test cases based on historical data. Requirements are a significant factor in the testing process, the priorities of test cases are initialized based on requirement priorities in our *history-based* approach, and then are calculated dynamically according to historical data in regression testing. To evaluate our approach, an empirical study on an industrial system is conducted. Experimental results show an improved performance for our proposed method using measurements of Average Percentage of Faults Detected and Fault Detection Rate.

CCS Concepts
Software and its engineering→Software testing and debugging

Keywords
Test Case Prioritization; Requirement Property; History Data; Regression Testing

1. INTRODUCTION
Software testing is an important and expensive process throughout the software development life cycle. Due to resource constraints [17], testers should make testing quality with minimal cost. Regression testing, the process of reusing test suites, is an effective method. Software testing ultimately tests whether the software system is consistent with the requirement. Therefore, associating test cases with requirement properties could improve testing efficiency.

Regression testing can generate large amounts of historical data that are continuously stored in a database without human caring about, and only a limited number of people could use these data to continue in-depth research. In general, most faults can be detected within the first-several times (especially the first time) of regression testing. When time and resources are limited, first-time sorting looks like more significant.

CSED'16,May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4157-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896941.2896949

In this paper, we present an initialization method for test case prioritization based on test cases and requirement properties relationship matrix. We also propose one heuristic *history−based* test case prioritization method. Finally, to evaluate our approach, an empirical study on an industrial system is conducted to demonstrate our method. The results of our study show that our *history-based* test case prioritization can greatly improve testing effectiveness. Overall, our technique contributes directly to improving the efficiency of regression testing.

In summary, the contributions of this work include:

- **Initialization.** We discuss the question whether initializing the order of test cases can improve the efficiency of regression testing, and define the initialization rule based on requirement classification and requirement importance.
- ***History-based* test case prioritization.** We propose a new approach for regression test case prioritization based on fault-detection history and define the prioritization algorithms by using the initialization variables. Finally, we present an empirical evaluation for our research questions.
- **Time constraint.** We also consider the time constraint in testing process and discuss the question whether history-based TCP approach can also improve test efficiency. A new evaluation metric-Fault Detection Rate is defined.

The rest of this paper is organized as follows. Section 2 describes the test case prioritization problem and related work. Section 3 presents the classification of requirement properties and proposes a method for initializing a test suite. Section 4 describes the heuristic *history-based* method in detail. Section 5 illustrates the experiment design and analysis. Finally, conclusions and future work are given in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 Test Case Prioritization Problem
Test Case Prioritization, using criteria, sorts test cases to detect the most faults as fast as possible in order to improve testing efficiency[11]. The complete definition of the TCP problem was first proposed by Rothermel et al. [1].

Given: T, a test suite already selected, PT, the set of all possible prioritizations (orderings) of T, and f, an objective function from PT to the real numbers which applied to any such ordering, yields an award value for that ordering.

Problem: Find T′ ∈ PT such that

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')].$$

2.2 Related work
The team of Rothermel and Elbaum et al was committed to, initially, TCP techniques mainly focusing on code-level testing [1,

2], such as statement coverage and transition coverage, in order to investigate the effectiveness of TCP techniques.

Zhang and Nie et al. [5] extended prioritization idea and proposed an algorithm called TCP_RP_TC that considered testing requirement priorities and test case costs. The prioritization technique must predict requirement priorities and test costs before test suite execution, but prediction was difficult in practice. Therefore, they asserted that one solution was to rely on test history information, but they failed to detail how to implement it. Kim et al. [4], which was closely related to ours, proposed a history-based TCP technique in which test cases were sorted according to historical values calculated from historical execution data. However, they did not consider requirement priorities. Yuchi et al. [6] presented a history-based cost-cognizant test case prioritization technique in regression testing. It inputted historical information of test cases into a genetic algorithm in order to produce an order. Chu-Ti et al. [9] presented a history-based TCP method with software version awareness.

There were other novel researches on test case prioritization. Arafeen and Do proposed a test case prioritization technique using requirements-based clustering [13]. In 2014, Elbaum and Rothermel et al presented techniques which made continuous integration processes more cost-effective [16]. In 2015, a new approach, REPiR, was introduced to address the problem of regression test prioritization by reducing it to a standard Information Retrieval problem such that the differences between two program versions form the query and the tests constitute the document collection [12].

In the work most closely related to ours, Srikanth et al. [3], proposed a value-driven approach for system-level test case prioritization, known as the Prioritization of Requirements for Test (PORT). However, they did not consider using history data to prioritize test cases. Previous studies on requirement factor or historical information have staged achievements [3-5, 6, 9], but they were limited to a single factor which has some limitation. One of the major breakthroughs in this paper is to consider two factors together and use them to prioritize test cases.

3. TCP INITIALIZATION

3.1 Classification of Requirement Properties

A series of requirement properties can be obtained in the requirements phase, and then developers implement these requirement properties by coding. Effective test cases are generated in these phases, and each test case is related to requirement properties. Each requirement property has its own priority so that it has a certain influence in TCP. We use Importance Value ($IV(r)$) to denote the priority of requirement property r. We classify the priority of requirement properties based on two factors.

Customer-assigned priority (CP) [3] is given by customers to indicate whether or not a requirement property is major. We subdivide the main requirement properties into three levels. The highest level properties involve system operation, such as exception handling and system warning. The higher level properties involve data input, output, and update. The rest main properties are considered as the lowest level. Similarly, non-main requirement properties can also be divided into two levels. The higher level properties involve data update and the lower level properties involve non data update. According to the level, a requirement property *r* is given an importance value from 5 to 1, denoted as *CP_IV(r)*. Table 1 shows the detailed classification and *CP_IV* assignment.

Developer-assigned priority (DP) is given by the developers to show the implementation complexity of each requirement property. This assignment is fully decided by developers. Five level standards are designated for DP. According to the level assigned by developers, a requirement property *r* is given an importance value, from 5 to 1, denoted as *DP_IV(r)*. The larger value means higher implementation complexity. In general, the requirement property with higher implementation complexity tends to have a greater number of potential faults, so the corresponding priority should be higher. Detailed classification and *DP_IV* assignment are given in Table 2.

Table 1. Classification and *CP_IV* assignment

Classification of CP	Level type	*CP_IV*
main	system-operation	5
	data-update	4
	non-data-update	3
non-main	data-update	2
	non-data-update	1

Table 2. Classification and *DP_IV* assignment

Level type	*DP_IV*
most-difficult-implementation	5
more-difficult-implementation	4
middle-difficult-implementation	3
simpler-implementation	2
simplest-implementation	1

According to Table 1 and Table 2, $IV(r)$ of a requirement property *r* can be calculated as following:

$$IV(r) = \omega_1 * CP\_IV(r) + \omega_2 * DP\_IV(r) \quad (1)$$

where ω_j (j = 1, 2) represents the weight for each factor. The sum of ω_j is 1. If we use a high value of ω_1, then the priority of the requirement property is increasingly dependent on the customers. Similarly, if a high value is assigned to ω_2, the priority of the requirement property is more dependent on the developers. $IV(r)$ is a measure of importance for a requirement property *r*. Generally speaking, ω_j would be assigned equal weights from experience if there is no special point. Actually, how to adjust ω_j to make it toward the favorable trend is also a valuable research.

3.2 TCP Initialization

Once the *IV* of every requirement property is determined, the relationship matrix can be obtained. Let $R = \{r_1, r_2, \cdots r_m\}$ be the set of requirement properties and $T = \{t_1, t_2, \cdots t_n\}$ a test suite. For each requirement $r \in R$, at least one test case *t* in *T* satisfies *r*. For each $t \in T$, there are zero or more requirement properties satisfying it. Therefore, *RP(t)* is the sum of *IV(r)*, which satisfies *t*. The satisfiability relation from *T* to *R*, denoted as $S(T, R)_{n\times m}$ and the algorithm of calculating *S* is as follows.

Algorithm 1 Calculate relationship *S*

```
1 : For each t_i ∈ T, r_j ∈ R
2 :     if (t_i satisfies r_j ) then
3 :         S(t_i, r_j) = IV(r_j);
4 :     else S(t_i, r_j) = 0;
5 :     endif
6 : endfor
```

Differing from the 0-1 matrix described in [7], *S* is that matrix in which the elements are real numbers. The prioritization value of a test case *t*, denoted by $RP(t)$, can be calculated by summing the *IV* of properties which *t* satisfies.

$$RP(t_i) = \sum_{j=1}^{m} S(t_i, r_j) \quad (2)$$

4. HISTORY-BASED PRIORITIZATION

4.1 Dynamic adjustment of RP prioritization

We first take into account of dynamic adjustment of IV. After testing $t \in T$, faults detected are recorded and are distributed to properties satisfied by *t*, indicating that how many faults each requirement property involves. If the number of faults is less than the previous one, $IV(r)$ is reduced by their difference, otherwise, $IV(r)$ is added by the difference. Therefore, *RP(t)* must be recalculated and the test cases must be reordered. Equation 3 denotes the adjustment calculation, where $IV(t_i, r_j)$ is the $IV(r_j)$ that t_i satisfies, $IV_n(t_i, r_j)$ is the current value, and $IV_{n-1}(t_i, r_j)$ is the last record in regression testing. $Dvalue\_fault$ is the difference value of detected faults between two adjacent records.

$$\mathrm{IV_n(t_i, r_j)} = \mathrm{IV_{n\text{-}1}(t_i, r_j)} + \mathrm{Dvalue\_fault} \quad (3)$$

We employ *RP* to achieve prioritization. Two methods are available for dynamic adjustment of RP: ***Total adjusted RP prioritization*** and ***Additional adjusted RP prioritization***. *Total adjusted RP prioritization* is defined to prioritize test cases according to descending order of RP values so that the test case with higher RP could be executed earlier. If several test cases have the same highest RP, we randomly select one of those test cases.

The other method named *Additional adjusted RP prioritization* is to dynamically adjust RP of remaining test cases after each selection of the best test case. However, the possibility of detecting new faults decreases when executing the rest of test cases that cover the same requirement property with selected test cases. Therefore, we define an *additional adjusted RP prioritization* as follows.

Let $\mathrm{Req(t_i)} = \{\mathrm{r} | \mathrm{S(t_i, r)} > 0\}$, and $|\mathrm{Req(t_i)}|$ be the number of elements in set $\mathrm{Req(t_i)}$. After the execution of test case t_i, the RP value of the unselected test case t_j is reduced by $|\mathrm{Req}(t_i) \cap \mathrm{Req}(t_j)|$. For each selection of the best test case, the RP of test cases may change; therefore, the test case with the highest RP in the current session should be selected.

Algorithm 2 Additional adjusted RP prioritization

// assume that the kth is current regression testing.

Input:
$T = \{t_1, t_2, \cdots t_n\}$, $R = \{r_1, r_2, \cdots r_m\}$,
$IV_{k-1}(t_i, r_j)$: IV values of requirement properties satisfied by $\mathrm{t_i}$ in last test cycle,
$fault_{k-1}(t_i, r_j)$: number of detected faults in last regression testing,
$fault_{k-2}(t_i, r_j)$: number of detected faults in last two regression testing,
$RP[n]$: set of RP value for each test case,
$Req[n]$: set of requirements satisfied by each test case
Output:
T′: the new test case sequence;

1. **for each** $t_i \in T$, $r_j \in R$
2. *Dvalue_fault* = $fault_{k-1}(t_i, r_j) - fault_{k-2}(t_i, r_j)$;
3. $IV_k(t_i, r_j) = IV_{k-1}(t_i, r_j)$ + *Dvalue_fault;*
4. **endfor**
5. **for each** $t_i \in T, r_j \in R$
6. Calculate $RP(t_i)$ based on the new obtained $IV_k(t_i, r_j)$;
7. **endfor**
8. $T' = \emptyset$;
9. **while** ($T \neq \emptyset$) **do**
10. t_{best} := SelectBest(T, $RP[n]$);
11. $T' = T' + \{t_{best}\}$;
12. $T = T - \{t_{best}\}$;
13. **for each** $t_i \in T$
14. $RP(t_i)$ = AdditionalStrategy ($Req(t_i)$,$Req(t_{best})$, $RP(t_i)$);
15. **endfor**
16. **endwhile**

Algorithm 2 is *Additional adjusted RP prioritization* method. Lines 1-4 show the method of $IV(t_i, r_j)$ adjustment based on the difference of the number of detected faults in adjacent regression testing. After adjusting *IV*, we can recalculate RP of each test case. Lines 8-16 are *Additional adjusted RP prioritization* and its two sub-functions named ***SelectBest*** and ***AdditionalStrategy***. The time required in Line1-4 to adjust $IV(t_i, r_j)$ is O(mn), and the time required in Line5-7 to recalculate $RP(t_i)$ is O(mn). Obviously, the time required in Line8-16 is $O(n^2)$ depending on the time complexity of Algorithm 3 and Algorithm 4.Therefore, the overall worst time complexity is $O(mn+ n^2)$.

Algorithm 3 SelectBest

Input:
$T = \{t_1, t_2, \cdots t_n\}$, $RP[n]$
Output:
t_{best}

1. Max_RP = 0;
2. **for each** t_i
3. **if** ($RP(t_i)$ > Max_RP) **then**
4. Max_RP = $RP(t_i)$;
5. candidateCount = 0;
6. best = i;
7. **endif**
8. **else if** ($RP(t_i)$ == Max_RP) **then**
9. candidateBest[candidateCount] = i;
10. candidateCount + +;
11. **endif**
12. **endfor**
13. **if**(candidateCount > 0) **then**
14. best = Random(candidateBest[candidateCount]);
15. **endif**

Algorithm 3 is *SelectBest* which select optimum solution in each cycle based on Greedy Algorithm. Prioritization is based on the descending order of RP. When the selection process meets two or more identical highest values, one test case must be randomly chosen as the current best. The worst time complexity of *SelectBest* is O(n).

Algorithm 4 AdditionalStrategy

Input:
$T = \{t_1, t_2, \cdots t_n\}$, $Req(t_i)$,$Req(t_j)$, $RP(t_i)$
Output:
$RP(t_i)$

1. **if** (t_i and t_j satisfied the same requirement properties and $t_i \neq t_j$) **then**
2. $RP(t_i) = RP(t_i) - \left|\boldsymbol{Req(t_i)} \cap \boldsymbol{Req(t_j)}\right|$;

The difference between *Total adjusted RP prioritization* and *Additional adjusted RP prioritization* is the sub-function *AdditionalStrategy* in Algorithm 4 which is called by *Additional adjusted RP prioritization* but not *Total adjusted RP prioritization*. So the algorithm of *Total adjusted RP prioritization* won't be tired in words here. According to *AdditionalStrategy*, RP values of remaining test cases may be changed in each selection. The worst time complexity of *AdditionalStrategy* is O(1).

4.2 History-based TCP framework

Each test case will be assigned a probability using historical data in our method. Because of limited time and money, all test cases cannot always be executed in each testing session. Therefore, we select the test cases with the highest probabilities. Since the probability calculation equation we define involves RP value, we normalize the RP as *NRP*:

$$\mathrm{NRP}(t_i) = \frac{RP(t_i)}{\sum_{j=1}^{n} RP(t_j)}. \quad (4)$$

The probability calculation is defined as follows:

$$\begin{cases} P_1 = \mathrm{NRP}_1 \\ P_k = \alpha \mathrm{NRP}_k + (1\text{-}\alpha)P_{k-1} \quad 0 \le \alpha \le 1,\ k \ge 1 \end{cases}. \quad (5)$$

P_k refers to the *k*-th probability for each test case executed, and initialization P_1 is the first *NRP*. The higher the value of P_k, the larger the executing probability of test case *t*. α is a smoothing constant used to weigh individual history RP values, and the tester can assign it according to the actual circumstance. If α is used as a high value, then the probability is primarily dependent on the change between the last two test sessions. Otherwise, history data of the entire phase (from first session to current session) are the main basis. Fig. 1 is a framework of the *history-based* test case prioritization technique.

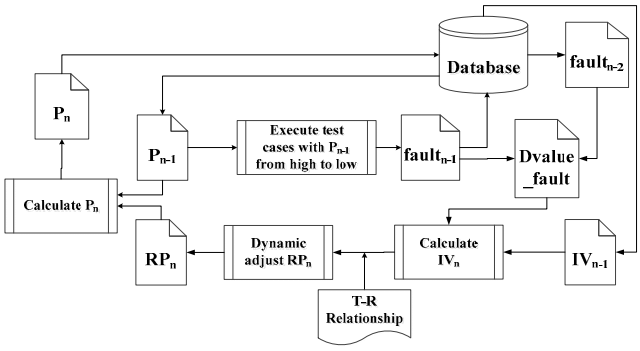

Figure 1. Framework of *History-based* Test Case Prioritization

In Fig. 1, the document icon denotes the input/output document of every execution action. For example, P_{n-1} document icon represents the probabilities of all test cases in *n*-1th regression testing obtained from the database. The rectangle icon means operation, specifically actions, of which most contains algorithms or equations. The cylinder icon denotes the database that stores the data of each testing session, such as the ordering of test cases and the number of detected faults. The rectangle with the curved edge represents the relationship between test cases and requirement properties.

Data flow can be determined from Fig. 1. Each time of regression testing, we first obtain all test cases with probabilities values from the database and then selected *n*% of them with the highest probabilities to execute, thereby allowing detected faults to be obtained. Then IV_n can be calculated by IV_{n-1} and the difference value of faults detected between two adjacent records. According to the *T-R* relationship, we dynamically adjust $\mathrm{RP_n}$. The *n*th RP values can also be obtained. Finally, P_n for each test case can be calculated based on Equation 5 and the P_n can be stored in the database as history information for future usage.

5. EMPIRICAL EVALUATION

To investigate the effectiveness of our *history-based* test case prioritization technique, we perform an empirical evaluation in terms of the following research questions.

- RQ1: Whether using our initialization method for the first regression testing improves testing efficiency?

We know that a vast majority of researches have not focused on first-time ordering, and initialization of test cases have been randomly generated generally. This research question aims to understand whether our initialization method have more effective than random initialization.

- RQ2: Is our *history-based* method more effective than other existing test case prioritization techniques?

Our second research question considers whether our *history-based* method can detect faults earlier than other existing traditional test case prioritization techniques, such as random prioritization and *RP-based* prioritization.

- RQ3: If there is time constraint, whether our *history-based* method can still improve testing efficiency?

We assume that only *n*% (20% or 50%) of all test cases can be executed because of limited time. In this research question, whether our *history-based* method can still have higher rate of faults detection.

5.1 Subject System

To perform the empirical evaluation, we use an industrial system called *CPMISS*(http://www.cpmiss.net/CPMISS/) for our experiments. It is a Web application for community services with approximately 440,000 lines of Java code (LOC). We use its 5 regression testing versions for experiments which from version 2.1-2.5. There are more than 500 atomic test cases generated by test team for black-box testing. According to the requirements specification, 71 requirement properties are summarized.

After completing requirement specifications, the tester asks the customers to indicate main requirement properties and non-main requirement properties, allowing us to assign the *CP_IV*. When the software is implemented, the tester requires developers to assign the *DP_IV*; equal weight is given for each factor. When we get these original data of the system, we first extract the T-R relationship as a matrix. Then calculate *IV* value for each requirement property and continue to calculate RP value for each test case according to T-R relationship matrix. The useful data is collected for our evaluation to the first research question.

Then, we run the algorithm 2-4 for each testing cycle and calculate probability P_k (current cycle is *k*th cycle.) for each test case according to Equation 4 and Equation 5. At last, the useful data is collected for evaluation analysis.

5.2 Prioritization strategies

Here we briefly describe four test case prioritization strategies for empirical comparison.

TCP initialization is our proposed initialization method which is described in Section III in detail. It initialize test-cases-ordering based on the importance of requirement properties. We set the weight $\omega_1 = 0.5$ and $\omega_2 = 0.5$.

Random prioritization sorts test cases randomly. This method has no technical actions and it is the simplest prioritization. Since random strategy is non-parametric, we applied random prioritization multiple times for each experiment [15].

***RP-based* prioritization** sorts test cases according to the descending order of RP (requirement priority) values [3, 14] so that the test case with higher RP could be executed earlier.

***History-based* prioritization** is our proposed method which is described in Section IV in detail. It can dynamically adjust the order of test cases after each test cycle according to historical fault information. The smoothing constant α is assigned 0.8 in order to take into account the change between the last two test sessions.

5.3 Evaluation Metrics

5.3.1 APFD

APFD [2], the weighted average of percentage of faults detected, focuses on the rate of fault detection over the life of a test suite. The higher the value, the earlier detect faults during the testing process. APFD is based on the assumption that two orders (faults and costs) are equivalent. The formula of APFD is as follows:

$$APFD = 1 - \frac{TF_1+TF_2+\cdots+TF_m}{nm} + \frac{1}{2n} \quad (6)$$

where *n* is the number of test cases and *m* is the number of faults. TF_i is the number of first test case in the execution order *T* that reveals fault *i*.

5.3.2 Fault Detection Rate

Because time constraint in our experiment is considered, we could not always execute all the test cases. FDR denotes the ability to detect faults in certain periods of execution time. We give its definition as follows:

$$FDR = \frac{FDT}{O\_FDT}, \quad (7)$$

where *FDT* is the number of detected faults by the current prioritization technique during a certain execution time and *O_FDT* is the optimal prioritization technique which sorts test cases according to the number of faults. The higher the value of FDR, the greater the ability of detecting faults of the prioritization technique during execution time.

5.4 Results and Analysis

In this section, we present the results of the experiments and analyze them relevant to our research questions above.

5.4.1 RQ1:test efficiency comparison between TCP initialization and random initialization

Our first research question considers whether it can improve the fault-detection abilities when using our TCP initialization method for the first regression testing. We conjectures that differences in initialization method will cause significant differences in fault detection, so we designs two experiments to respond to this question: Experiment1a involving TCP initialization technique and Experiment1b involving random initialization technique. Since random initialization is non-parametric, we run it 20 times for this research question. The presented result is the average of 20 data. Both techniques are run in the same subject system and same version (version 2.1).

Table 3. APFD for initialization

Experiment	Initialization	APFD (%)
1a	TCP method	55.48
1b	random method	40.89

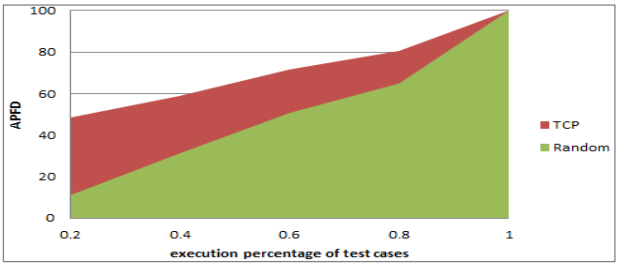

Figure 2.APFD for initialization

Fig. 2 illustrates the relationship between execution percentage of test cases and percentage of fault detected for two techniques. As shown in Fig. 2, our TCP initialization can more quickly detect faults than random initialization. In addition, as shown in Table 3, APFD of the test sequence is higher than random initialization by 55.48%. Therefore, for regression testing process, the first-time ordering of test cases generated based on requirement properties has greater efficiency than ordering generated randomly. Our initialization method can effectively save time and resources.

Even though initialization consumes some time, it seems that the random can save testing time. However, it is not true. The calculation information in initialization is used in the later regression testing. If you randomly sort test cases in initialization, you still have calculation that information in later cycle when using our history-based method for test case prioritization. Time cost is equal with TCP initialization for the global testing. So, TCP initialization is superior to random initialization.

5.4.2 RQ2:test efficiency compared with other existing test case prioritization for regression testing

Our second research question considers whether *history-based* prioritization technique can improve test efficiency compared to other existing test case prioritization techniques. To address this research question we perform three experiments: Experiment2a involving random test case prioritization technique; Experiment2b involving *RP-based* test case prioritization technique; and Experiment2c involving *history-based* test case prioritization technique. We use the same experiment design, but system versions are different, choosing version2.1 to 2.5, five times regression testing. In addition, we also run our *history-based* method 20 times for each cycle because of its non-parametric in several algorithms.

Fig. 3 is a scatter plot showing APFD distribution of each TCP technique for five-times regression testing. The horizontal axis indicates five versions (2.1-2.5) and the vertical axis indicates APFD value. Each legend represents a kind of TCP technique: the rhombus legend denotes random prioritization; the square represents RP-based prioritization; and the circle is *history-based* prioritization. From Fig.3, we can see that *history-based* prioritization has higher APFD values than random technique in all of versions and from version2.2 our *history-based* method is also better than *RP-based* method.

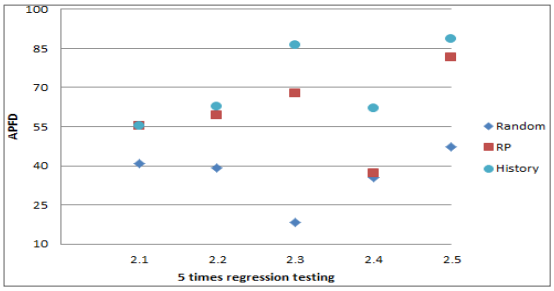

Figure 3.APFD for regression.

Table 4 shows the APFD of each regression testing for each TCP technique, including each median, average, and standard deviation. From Table 4, we can see that our *history-based* prioritization technique is more effective than other techniques. From version2.1, APFD values of *history-based* prioritization are higher than random prioritization and *RP-based* prioritization and its APFD value reaches 89.54% in version2.5. The higher APFD value is, the more faults can be detected in the earlier time. Due to the randomness of selection process, the effectiveness of random prioritization may fluctuate and it is usually below the average. It is not stable. In our experiments, random prioritization had the lowest APFD.

We use the standard *one-tail t-test* to check that whether measure results can verify their effectiveness. We assume that f_1 and f_2 are defined as the value of APFD and prioritized by two test case prioritization methods, respectively. The following two hypotheses are considered:

H_0: $f_1 = f_2$, if two techniques have the same effectiveness of fault detection.

H_1: $f_1 > f_2$, if f_1 is significantly better than f_2.

If the *p*-value is less than the significance level ($\alpha = 0.05$), our results prove to have significantly reliability.

We use the data of five system versions for statistical analysis and the result is shown as Table 5. According to Table 5, *history-based* prioritization is much better than the random prioritization and *RP-based* prioritization, because its value of *t* statistic is more than 0 and the *p*-value is less than 0.05, demonstrating a significant difference between *history-based* prioritization and the other methods.

Table 5. Statistical analyses of APFD

TCP Techniques	*t* Stat	*p*-value (one-tail)
History-based vs Random	3.7267	0.0102
History-based vs *RP-based*	2.5882	0.0304

***History-based* vs. Random.** The *t*-value is 3.7267 ($t>0$) and the *p*-value is 0.0102 ($p<0.05$) from Table 5. The values show that *History-based* prioritization demonstrates greater efficiency when detecting faults.

***History-based* vs. *RP-based*.** The *t*-value is 2.5882 ($t>0$) and the *p*-value is 0.0304 ($p<0.05$) from statistical analyses of APFD, meaning that significant difference is observed between the two methods. Therefore, *History-based* prioritization is better than *RP-based* prioritization.

5.4.3 RQ3: time constraint in regression testing

Our third research question considers whether *history-based* method can still improve testing efficiency when there is a time constraint in regression testing. We assume that only *n*% of all test cases can be executed because of limited time. We conjecture that differences of *n*% values will cause significant differences in fault detection, so we design six experiments to respond to this question: Experiment3a, 3b and 3c involving 20% of test cases executed and Experiment3d, 3e and 3f involving 50% of test cases executed. It is similar to those performed to address RQ2: we use the same experiment design, only increasing time constraint.

Table 6 shows the FDR values of each regression testing for each TCP technique, including each median, average, and standard deviation when executing 20% or 50% of test cases. When executing 20% of test cases, the median value of *history-based* prioritization is 78.31%, and the average is 82.986%, which is much higher than the other techniques. The FDR values of *history-based* prioritization tend to 100% in the first three times of regression testing. We also find that in the fourth regression testing, the FDR value declines to 78.31% because the number of faults increases and new faults are introduced. Random prioritization is the weakest method because its median and average are the lowest. When executing 50% of test cases, the median and average for *history-based* prioritization are the highest than the others and the FDR values of *history-based* prioritization and *RP-based* prioritization are 100% in the fifth regression testing. That is because the fault number declines to single-digit levels from the third regression testing and after two prioritization, we can detect nearly all the faults in 50% of test cases.

5.4.4 Overall analysis

In conclusion, *History-based* prioritization is superior to random prioritization or the traditional one-time test case prioritization (*RP-based* prioritization). *History-based* prioritization is the best choice for regression testing when executing all test cases because it can detect faults as fast as possible. Even with time constraint and lack of sufficient time to execute all test cases, *history-based* prioritization is also a great method and demonstrated the greatest effectiveness.

5.5 Threats to Validity

5.5.1 Threats to internal validity

Threats to internal validity include the efficiency of requirements classification. To reduce this threat, we choose the commonly-used five-point scale for classification. Another threat to internal validity is the assignment of the weights ω_1 and ω_2. We should depend on reality condition. In general, the equal weights can introduce the least threat when no empirical research for the weights. So we set $\omega_1 = 0.5$ and $\omega_2 = 0.5$, to reduce this threat.

5.5.2 Threats to external validity

We use an industrial project as our experiment objective. The situation of fault occurrence may be different in other projects

Table 4. APFD and its median, average, and standard deviation of several TCP techniques (%)

Experiment	TCP techniques	versions							
		2.1	2.2	2.3	2.4	2.5	Median	Average	Standard Deviation
2a	Random	40.89	39.09	18.24	35.38	47.39	39.09	36.198	9.786
2b	*RP-based*	55.48	56.36	67.65	37.17	81.70	56.36	59.672	14.723
2c	*History-based*	55.48	62.84	86.47	61.94	89.54	62.84	71.254	13.945

Table 6. FDR and its median, average, and standard deviation of several TCP techniques (%)

Experiment	*n%*	TCP techniques	versions							
			2.1	2.2	2.3	2.4	2.5	Median	Average	Standard Deviation
3a	20%	Random	18.07	8.330	0	9.090	16.67	9.09	10.432	6.516
3b		RP-based	78.31	41.67	60.00	18.18	50.00	50.00	49.632	19.913
3c		History-based	78.31	63.89	100.0	72.73	100.0	78.31	82.986	14.633
3d	50%	Random	41.88	32.08	0	36.36	16.67	32.08	25.398	15.216
3e		RP-based	76.92	64.15	80.00	63.64	100.0	76.92	76.942	13.279
3f		History-based	76.92	62.26	100.0	72.73	100.0	76.92	82.382	15.157
Number of Faults			102	53	5	11	6			

which we have not used. However, the fault occurrence is real in industrial projects, not we manually seeding, so we think our experiment can be representative a series of situations.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a *history-based* test case prioritization and give an initialization rule for the first regression testing. As faults are naturally-occurring and the distribution feature is not evenly distributed, our initialization method demonstrates better efficiency than random method. For regression testing, experimental results show that our *history-based* TCP method has the best effectiveness and fault-detection ability with significant difference compared with the other methods.

However, our proposed method has some shortcomings. There may be redundant faults when subdividing faults to every requirement property. In the future, redundancy elimination may be the first consideration. We intend to extend our experiment to diversified programs such as programs with faults of different distributions or larger scales.

7. ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (NSFC) under grant No. 61572306.

8. REFERENCES

[1] Elbaum, S., Malishevsky, A. G. and Rothermel, G. 2000. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*. 102-112.

[2] Elbaum, S., Malishevsky, A. G., and Rothermel, G. 2002. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 159-182.

[3] Srikanth, H. and Banerjee, S. 2012. Improving test efficiency through system test prioritization. *J. Syst. Softw.* 85, 5 (May. 2012), 1176-1187.

[4] Kim, J. M. and Porter, A. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. *In Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*. 119-129.

[5] Zhang, X., Nie, C., Xu, B., and Qu, B. 2007. Test case prioritization based on varying testing requirement priorities and test case costs. *In Proceedings of the 7th International Conference on Quality Software (QSIC'07)*. 15–24.

[6] Huang, Y. C., Peng, K. L., and Huang, C. Y. 2012. A history-based cost-cognizant test case prioritization technique in regression testing. *J. Syst. Softw.* 85, 3 (Mar. 2012), 626-637.

[7] Wang, X. and Zeng, H. 2014. Dynamic test case prioritization based on multi-objective. *In Proceedings of the 15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'14)*. 1-6.

[8] Rothermel, G. and Harrold, M. J. 1996. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.* 22, 8 (Aug. 1996), 529-551.

[9] Lin, C. T., Chen, C. D., Tsai, C. S. and Kapfhammer, G. M. 2013. History-based test case prioritization with software version awareness. *In Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS'13)*. 171-172.

[10] Marijan, D., Gotlieb, A., and Sen, S. 2013. Test case prioritization for continuous regression testing: An industrial case study *In Proceedings of the 29th International Conference on Software Maintenance (ICSM'13)*. 540-543.

[11] Yoo, S. and Harman, M. 2012. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.* 22, 2 (Feb. 2012), 67-120.

[12] Saha, R. K, Zhang, L., Khurshid, S., and Perry, D. E. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. *In Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. 268-279.

[13] Arafeen, M. J., and Do, H. 2013. Test case prioritization using requirements-based clustering. *In Proceedings of the 6th International Conference on Software Testing, Verification, and Validation (ICST'13)*. 312-321.

[14] Li, S., Bian, N., Chen, Z., You, D., and He, Y. 2010. A simulation study on some search algorithms for regression test case prioritization. *In Proceedings of the 10th International Conference on Quality Software (QSIC'10)*. 72-81.

[15] Arcuri, A, and Briand, L. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. *In Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. 1–10.

[16] Elbaum, S., Rothermel, G., and Penix, J. 2014.Techniques for improving regression testing in continuous integration development environments. *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235-245.

[17] Qian, H and Andresen, D. 2016. Automate Scientific Workflow Execution between Local Cluster and Cloud. *IJNDC*. 4, 1 (Jan. 2016), 45-54.