Software Qual J (2014) 22:335–361
DOI 10.1007/s11219-013-9224-0

Similarity-based test case prioritization using ordered sequences of program entities

Chunrong Fang · Zhenyu Chen · Kun Wu · Zhihong Zhao

Published online: 21 November 2013
© Springer Science+Business Media New York 2013

Abstract Test suites often grow very large over many releases, such that it is impractical to re-execute all test cases within limited resources. Test case prioritization rearranges test cases to improve the effectiveness of testing. Code coverage has been widely used as criteria in test case prioritization. However, the simple way may not reveal some bugs, such that the fault detection rate decreases. In this paper, we use the ordered sequences of program entities to improve the effectiveness of test case prioritization. The execution frequency profiles of test cases are collected and transformed into the ordered sequences. We propose several novel similarity-based test case prioritization techniques based on the edit distances of ordered sequences. An empirical study of five open source programs was conducted. The experimental results show that our techniques can significantly increase the fault detection rate and be effective in detecting faults in loops. Moreover, our techniques are more cost-effective than the existing techniques.

Keywords Test case prioritization · Similarity · Ordered sequence · Edit distance · Farthest-first algorithm

1 Introduction

Test suites often contain tens of thousands or even more test cases for large software. The cost of executing all the test cases is huge. The situation is serious especially for regression

A preliminary version of this paper was presented at the 7th International Workshop on Automation of Software Test (Wu et al. 2012).

C. Fang · Z. Chen · K. Wu · Z. Zhao
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China

C. Fang · Z. Chen (✉) · K. Wu · Z. Zhao
Software Institute, Nanjing University, Nanjing 210093, China
e-mail: zychen@software.nju.edu.cn

testing. It is impractical to re-execute all test cases within limited resources, since test suites often grow very large over many releases. Many techniques have been proposed to solve this problem at least partially, one of which is test case prioritization (Rothermel et al. 2001). Test case prioritization rearranges test cases according to certain criteria, such that the test cases with higher priority are executed earlier (Elbaum et al. 2002). Previous studies have shown that test case prioritization can be cost-effective in many cases (Elbaum et al. 2002; Malishevsky et al. 2002).

Test case prioritization techniques can be used in many scenarios. In this paper. we focus on the regression testing scenario. Therefore, two cases need to be considered: general test case prioritization and version-specific test case prioritization (Rothermel et al. 2001). The former is applied to a base version of the software under test (SUT) with no knowledge about the modifications to it, in the hope that the prioritized test suite will be effective over the subsequent versions. The latter, on the other hand, is performed after a set of changes have been made to the current version and is concerned only with that particular version. In our paper, we study the case of general test case prioritization.

The purpose of test case prioritization is to increase the probability that test cases meet certain objectives when executed in a specific order. The main objective of the test case prioritization is to increase the fault detection rate (i.e., revealing faults as early as possible). However, it is difficult to discover the fault detection information until the testing is finished. Hence, in practice, test case prioritization techniques rely on surrogates, hoping that early satisfying of these surrogates will lead to increasing the fault detection rate (Yoo and Harman 2012). Utilizing code coverage information as surrogates, coverage-based test case prioritization rearranges test cases in order to maximize code coverage as early as possible (Elbaum et al. 2002; Rothermel et al. 2001). However, code coverage is not sufficient to guarantee a high fault detection rate in some cases (Marick 1994).

Similarity-based techniques have been introduced to utilize the execution profiles of test cases (Dickinson et al. 2001; Hemmati and Briand 2010; Jiang et al. 2009; Yan et al. 2010; Zhang et al. 2010). The purpose of similarity-based techniques is to maximize the diversity (i.e., minimize the similarity) of selected test cases. The diversity of test cases is computed by a certain dissimilarity measure between each pair of test cases. Consequently, this will increase the chance of detecting faults as early as possible if we maximize the diversity of the test cases. Similarity-based techniques are mainly classified into two types: distribution-based and adaptive random testing (ART)-inspired. Distribution-based techniques cluster test cases according to their dissimilarities. Clusters and isolated points can be used to guide test case selection and test case prioritization. The intuition behind the idea is that the test cases that find different faults belong to different clusters based on the (dis)similarity measures (Chen et al. 2011; Hemmati et al. 2010; Zhang et al. 2010). ART-inspired test case prioritization is an extension of ART. Each time, a candidate set full of not-yet-selected test cases is constructed and the test case (from the candidate set) farthest away from the prioritized test suite is selected as the next one (Jiang et al. 2009).

Previous studies directly used either the code coverage information (Jiang et al. 2009) or the actual values of the execution counts in the similarity measures (Leon and Podgurski 2003). To the best of our knowledge, none of them considered the relative execution frequencies of program entities in test case prioritization. However, some difficult-to-find bugs are often caused by interactions between multiple program entities (Lo et al. 2009). These bugs, to a certain extent, may be sensitive to the relative execution counts of program entities other than the actual counts themselves. For example, long runs may show a similar behavior to shorter runs (e.g., executing a loop 1,000 times in one example and 10 in another).

In this paper, we provide a deep insight into the idea of similarity-based test case prioritization using the frequency profiles of test cases. We characterize each frequency profile as the ordered sequence of its entities, where the sorting key is the number of times each entity is executed. The distance of test cases is measured by the edit distance of ordered sequences. Our test case prioritization techniques rearrange test cases to maximize the diversity based on the distances of test cases. The ordered sequence of program entities has been used in fault localization and has been shown to be effective (Renieres and Reiss 2003). In this paper, we introduce it into test case prioritization for the first time, and the experimental results show that our techniques can improve the effectiveness of test case prioritization.

In summary, this paper mainly makes the following contributions:

(1) To the best of our knowledge, this is the first time that the ordered sequence of program entities measured by execution frequency is used in test case prioritization.
(2) We propose several novel similarity-based test case prioritization techniques based on the farthest-first ordered sequence (FOS) algorithm and greed-aided-clustering ordered sequence (GOS) algorithm, respectively.
(3) An empirical study is conducted and the experimental results show that our techniques can be more effective and cost-beneficial in increasing the fault detection rate.

The rest of the paper is organized as follows. Section 2 describes the background and related work. Section 3 illustrates a motivating example. Section 4 gives a detailed description of our techniques. Section 5 discusses the empirical study, and the results are analyzed in Sect. 6. Section 7 outlines the conclusion and future work.

2 Background and related work

The test case prioritization problem (Rothermel et al. 2001) can be formally defined as: given a test suite T, a set O containing all permutations of T, and a function f from O to the real numbers, find an $o \in O$ such that $(\forall o' \in O)[f(o) \geq f(o')]$. In this paper, the specific purpose is to find an order o of test suite T aimed to increase the likelihood of detecting faults in the early stages of the testing process. To qualify the fault detection rate of a given test suite, f is always an average percentage of the faults detected (APFD) function. APFD values range from 0 to 1, and a higher APFD score implies a higher fault detection rate. However, in practice, the fault detection information cannot be obtained completely until testing is finished. Test case prioritization techniques usually incorporate some surrogates to approximate fault detection. The ways of using surrogates are classified into two types: coverage-based techniques and similarity-based techniques.

2.1 Coverage-based techniques

Rothermel et al. (2001) first proposed a series of coverage-based test case prioritization techniques incorporating statement coverage and branch coverage. Elbaum et al. (2002) focused on function coverage-based prioritization techniques. In their studies, two algorithms, which are the most widely used, are involved: the total greedy algorithm and the additional greedy algorithm. These algorithms rearrange test cases in order to maximize code coverage as early as possible (Elbaum et al. 2002). However, it is known that these two algorithms may produce suboptimal results because they might construct results

denoting only local minima within the search space (Rothermel et al. 2001). Li et al. (2007) investigated some other greedy algorithms, such as the 2-optimal algorithm, hill-climbing algorithm, genetic algorithm, and so on. Although a weakness of the local optimum can be overcome, these algorithms cannot significantly outperform the greedy algorithms (Elbaum et al. 2002), which are considered as the "the cheaper-to-implement-and-execute" algorithms. In this paper, two greedy algorithms are selected as peer ones.

Jones and Harrold (2003) provided insights into test case prioritization on MC/DC (modified condition/decision coverage). They presented new algorithms for test case prioritization effectively incorporating MC/DC. Fang et al. (2012) investigated the fault section capability of a series of logic coverage criteria on test case prioritization, such as MC/DC and several fault-based logic coverages. Do et al. (2004) applied prioritization techniques to object-oriented systems in a JUnit testing environment, and these techniques adopted block-level and method-level coverages. Do et al. (2010) investigated the effects of time constraints for test case prioritization based on an economic model.

Comparing with these coverage-based techniques, we aim at similarity-based techniques in this paper.

2.2 Similarity-based techniques

Similarity-based techniques are introduced to maximize the diversity of test cases. Jiang et al. (2009) proposed the first set of coverage-based ART techniques in order to spread test cases as evenly as possible across the code coverage space. Yoo et al. (2009) combined clustering with expert knowledge. However, the goal of this technique is to incorporate human efforts into the prioritization process, which is different from ours. Leon and Podgurski (2003) proposed two methods of distribution techniques: cluster filtering and failure-pursuit sampling. Carlson et al. (2011) showed that test case prioritization that utilizes a clustering approach can improve the effectiveness of the test case prioritization technique. Masri et al. (2007) compared these techniques based on exercising various types of complicated information flows. Hemmati et al. (2010) and Hemmati and Briand (2010) proposed similarity-based techniques to reduce the cost of model-based test case selection. Yan et al. (2010) proposed a new cluster sampling strategy called execution-spectra-based sampling to improve cluster filtering. Zhang et al. (2010) introduced a new regression test selection technique to trade-off between test suite reduction and fault detection capability. Chen et al. (2011) used semi-supervised learning to improve clustering, and also used program slicing to improve the effectiveness of cluster test selection.

The successful stories of similarity-based techniques inspired us to use even more execution information to improve test case prioritization. In this paper, we propose several novel test case prioritization techniques based on the ordered sequences of program entities, which are widely used in debugging. Renieres and Reiss (2003) used ordered sequences to select a successful run that is as similar as possible to the faulty run. In Renieres and Reiss (2003), with the help of relevant execution counts of the runs, the components that execute the same numbers of times would appear different for any run of even minuscule length differences. In addition, a distance metric, which is similar to the edit distance, was adopted to assist peer selection (Sumner et al. 2011).

Sampath et al. (2008) proposed a frequency-based prioritization technique applied to Web applications. It is assumed that faults will exist in the application code. Thus, they considered only sequences between pages that access application code, such that the fault detection rate would be improved by selecting test cases covering the most frequent sequences. In addition, this technique selects some features of Web application profiles,

which is the page access sequence of size 2, while not focusing on all the execution sequences of the whole application. Furthermore, this technique directly adopts a random algorithm to select test cases containing the most frequent page access sequences. Overall, this technique is different from ours.

Comparing with traditional similarity-based techniques, we first introduce the relative execution frequencies of program entities into test case prioritization and propose two algorithms, FOS and GOS, based on the existing algorithms.

3 Motivating example

This section illustrates a motivating example of how the ordered sequence technique can be effective in increasing the fault detection rate. Table 1 shows the program which calculates the quotient of a parameter divided by 2. The program has one fault in statement 5. There are four test cases: variable "a" is assigned to 0, 1, 2, and 3, respectively. t_4 can reveal the fault in statement 5. Therefore, the technique assigning t_4 to a higher priority will detect the fault earlier. In the table, the last three columns list the statement coverage information. The corresponding cell is assigned to n if the statement is covered n times by that test case. If a statement is not covered, then $n = 0$ and the corresponding cell is assigned to 0.

In most cases, t_3 will be run first, because it covers the largest number of statements. However, t_3 cannot detect the fault, even though it executes the faulty statement. The problem of test case prioritization is to select t_1(or t_2) or t_4 in the next step. Coverage-based techniques cannot distinguish t_1(or t_2) and t_4 because all statements are already covered. For similarity-based techniques, we should calculate pair-wise distances of these test cases in the next step.

The first case evaluates the statement coverage information of test cases by proportional binary distance, which is used in Dickinson et al. (2001). The proportional binary distance is an improved Euclidean distance to normalize the coverage information. And the formula of proportional binary distance between the two test cases can be found in Leon and Podgurski (2003). The proportional binary distance of (t_3, t_4) is 1.77 and that of (t_1, t_3)[or(t_2, t_3)] is 2.07. Therefore, t_1 (or t_2) is selected to maximize diversity. Then, the final test sequence will be t_3, t_1, t_4, and t_2, in which the positions of t_1 and t_2 can be exchanged.

The second case is the ordered sequence of statement execution counts. After sorting statements by their execution counts in ascending order, the ordered sequences of statements for each test case are <3, 4, 5, 1, 2, 6>, <3, 4, 5, 1, 2, 6>, <1, 3, 4, 5, 6, 2>, and <4, 1, 6, 3, 5, 2>. The edit distance[1] of (t_3, t_4) is 4 and that of (t_1, t_3)[or(t_2, t_3)] is 3. As a result, t_4 is selected to maximize diversity. Consequently, the final test sequence will be t_3, t_4, t_1, and t_2, or t_3, t_4, t_2, and t_1. Comparing these two techniques, we can discover that the ordered sequence technique executes t_4 earlier than the existing technique, that is, it can detect the fault earlier. At the same time, both the approaches can improve the rate of fault detection compared with the default order of the test cases.

In a passed run, Line 4 cannot be executed fewer times than the statements outside the loop, while in a failed run, the execution times of Line 4 must be fewer than the statements outside the loop. This would cause Line 4 to result in differently ordered sequence ranks, and our approach should catch this feature, resulting in a high rate of fault detection, that is, this bug is sensitive to the relative execution counts of program entities and can be captured earlier by our approach.

[1] http://en.wikipedia.org/wiki/Levenshtein_distance

Table 1 Motivating example

ID	Program	Test cases			
		t_1(a = 0)	t_2(a = 1)	t_3(a = 2)	t_4(a = 3)
	calculate(int a){				
1	int b = 0;	1	1	1	1
2	while(a > 1){	1	1	2	3
3	if(a%2 == 0)	0	0	1	2
4	b += 1;	0	0	1	0
5	a −= 2;}/*a −= 1*/	0	0	1	2
6	System.out.println(b); }	1	1	1	1
	Result	Pass	Pass	Pass	Fail

4 Approach

(Dis)similarity measures are the key parts of test case prioritization techniques. After determining what information should be evaluated and how this evaluation should be done, the last step is using a strategy to rearrange test cases in order to maximize diversity.

4.1 Ordered sequence of program entities

An execution profile is a characteristic summary of a program's behavior on a test case. In this paper, we compute a frequency profile that records the number of times each program entity (statement or branch) executed during the test.

Definition 1 (*Frequency Profile*) Given a program P with entities $<e_1, e_2, \ldots, e_n>$ and a test case t, the frequency profile with regard to t is $C(t) = <c_1, c_2, \ldots, c_n>$, in which c_i is the execution count of e_i during the test of t.

To characterize the discrimination of execution profiles, each profile of a test case is transformed into an ordered sequence of program entities, which is sorted by their execution frequencies.

Definition 2 (*Ordered Sequence*) Given a frequency profile $<c_1, c_2, \ldots, c_n>$ of a test case t, the ordered sequence of program entities with regard to t is $S(t) = <e_{i_1}, e_{i_2}, \ldots, e_{i_n}>$, in which $c_{i_1} \leq c_{i_2} \leq \cdots \leq c_{i_n}$.

Please note that some execution counts may be equal. To get a unique ordered sequence for a frequency profile, we assign the program entities with the original orders if their execution counts are equal, that is, if $c_{i\_j} = c_{i\_k}$ and $i_j < i_k$, then e_{i_j} is before e_{i_k} in $S(t)$.

In computer science, edit distance between the two strings of characters generally refers to the Levenshtein distance, which is defined as the smallest number of insertions, deletions, and substitutions required to change one string or tree into another (Wagner and Fischer 1974). We introduce edit distance to measure the distance between each pair of ordered sequences. The edit distance between the two ordered sequences is the cost of transforming one to the other based on a certain set of operations. Here, the operations allowed are insertion, deletion, and substitution. For example, the edit distance between "kitten" and "sitting" is 3: (1) substitution of "s" for "k;" (2) substitution of "i" for "e;", and (3) insertion of "g" at the end. On the one hand, these operations can capture the

phenomenon of executing the body of one loop more or fewer times than the body of another. On the other hand, these operations can attribute the differences between the two execution profiles to the basic entities involved in the editing operations (Renieres and Reiss 2003).

Definition 3 (*Distance*) The distance of two test cases t and t' is defined as the edit distance of their ordered sequences: $(t, t') = EditDistance(S(t), S(t'))$.

4.2 FOS: farthest-first ordered sequence

FOS algorithm is similar to the ART algorithm (Jiang et al. 2009). The main difference is that all the not-yet-selected test cases are included in the candidate set for FOS. For ART, the construction of the candidate set involves randomness and parts of the not-yet-selected test cases are included in it. FOS eliminates this randomness and provides more candidates, which may lead to a better choice.

FOS is described in Algorithm 1. The main procedure is PRIORITIZATION, which at first selects a test case that yields the greatest code coverage. Then, it repeatedly invokes the procedure SELECTNEXTTESTCASE to select an unordered test case into the prioritized set until all the test cases are reordered. The SELECTNEXTTESTCASE procedure constructs all the remaining not-yet-selected test cases as the candidate set and selects a test case that farthest away from the already selected ones. To decide which candidate test case is to be selected, two kinds of distances need to be considered: the first refers to the pair-wise distance between the test cases, while the second refers to the distance between a candidate test case and the already selected ones. For the former, we use edit distance to calculate it. For the latter, we first calculate all the distances between a candidate test case and each one in the selected set. Then, we choose the minimum distance as the representative for the distance between that candidate test case and the already selected ones.

Algorithm 1 FOS Algorithm

Input: T $\{t_1, t_2 ...\}$ is a set of test cases in original order
Output: T' $\{t'_1, t'_2 ...\}$ is a set of test cases in prioritized order

```
1: procedure PRIORITIZATION
2:     Set T' = ∅
3:     Select t_first with maximum coverage ratio from T
4:     T' = T' ∪ t_first, T = T \ t_first
5:     while T is not empty do
6:         Invoke SELECTNEXTTESTCASE(T, T')
7:     end while
8: return T'
9: end procedure
10:
11: procedure SELECTNEXTTESTCASE(T, T')
12:     Conduct candidate set C {c_1, c_2, ...}
13:     Conduct distance set D {d_1, d_2, ...}
14:     for all Test case c_i ∈ C do
15:         Set d_i = distance between c_i and P
16:     end for
17:     Select the maximum d_i from D
18:     T' = T' ∪ t_i, T = T \ t_i
19: end procedure
```

After calculating all the distances between a candidate test case and each one in the selected test set, there are at least three strategies that can be used to select one value as representative: minimum, average, and maximum. "Minimum" selects the minimal value, "maximum" selects the maximal value, and "average" calculates the mean value of all of them. All the three strategies are evaluated by measuring how quickly a test suite can detect faults (Jiang et al. 2009). The results indicated that the group using minimum strategy has the highest rate of fault detection. According to these results, we adopt the minimum strategy in our study.

4.3 GOS: greed-aided-clustering ordered sequence

For greed-aided-clustering process, test cases in each cluster are prioritized by the additional greedy algorithm and then the test cases are selected from each cluster according to the orderings.

GOS is shown in Algorithm 2: (1) form a cluster set C, in which each cluster has only one test case; (2) calculate the distances between each cluster and merge the two clusters with minimum distance; (3) repeat (2) until the size of C is less than n, which is a given input; (4) prioritize test cases in each cluster by additional greedy algorithm; (5) iteratively, select a test for each cluster c_i, add c_i into T', and if c_i is empty, remove c_i from C; and (6) repeat (5) until C is empty. For example, given a cluster C with three clusters $<<t_3, t_5>$, $<t_4, t_1>$, $<t_2>>$, they turn into $<<t_5, t_3>$, $<t_4, t_1>$, $<t_2>>$ after step (4) and the prioritized test cases would be $<t_5, t_4, t_2, t_3, t_1>$.

Algorithm 2 GOS Algorithm

Input: T $\{t_1, t_2, ...\}$ is a set of test cases in original order n: Cluster numbers
Output: T' $\{t'_1, t'_2, ...\}$ is a set of test cases in prioritized order

```
Set T' = ∅
Form |T| clusters: C{c1, c2...}
while |C| >= n do
    Find a pair of clusters (ci, cj) with minimum distance
    ck = ci ∪ cj
    C=C∪{ck}, C=C\ci,C=C\cj
end while
for all ci ∈ C do
    Prioritize ci according to additional greedy algorithm
end for
m = 1
while |C| > 0 do
    T' = T'∪{cm(1)}, cm=cm\cm(1)
    if |cm| = 0 then
        C=C\cm
    end if
    m = (m mod |C|) + 1
end while
return T'
```

4.4 Prioritization techniques

Based on the ordered sequences of program entities, we propose several novel test case prioritization techniques based on FOS and GOS, respectively. These techniques are also studied based on two levels of granularity of program entities: statement and branch. And

edit distance is used to measure distances between the test cases. Table 2 lists all our techniques (*T*7–*T*10) as well as other techniques (*T*1–*T*6) considered in our study. Details of Table 2 will be discussed in the next section. In our paper, re-st-ed-FOS represents a FOS technique utilizing the ordered sequences of statements measured by edit distance, re-br-ed-FOS represents a FOS technique utilizing the ordered sequences of branches measured by edit distance, re-st-ed-GOS represents a GOS technique utilizing the ordered sequences of statements measured by edit distance and re-br-ed-GOS represents a GOS technique utilizing the ordered sequences of branches measured by edit distance.

5 Experiment

5.1 Research questions

Our experiment aims to explore the following questions:

RQ1:Can our techniques increase the fault detection rate?
RQ2:Can our techniques reveal bugs in loops more quickly?
RQ3:Can these factors (granularity of coverage information and algorithm) have significant impacts on the fault detection rate?
RQ4:Can our techniques be more cost-effective than others?

5.2 Peer techniques for comparison

To investigate the effectiveness of the FOS and GOS techniques, we perform several empirical studies on open source programs. Random prioritization is ignored in our experiment, because it was not effective in previous studies (Elbaum et al. 2002; Wu et al. 2012). As listed in Table 2, the peer techniques for comparison are two coverage-based techniques and four similarity-based techniques. In Table 2, "orig" represents the execution profiles of program entities, while "re" represents relative execution counts of program entities. "st" stands for statement coverage and "br" stands for branch coverage. "ed" represents edit distance and "pb" represents proportional binary distance, while "addtl" represents additional greedy algorithm.

The additional statement prioritization technique (addtl-st) selects, in turn, the next test case which covers the maximum number of statements that are not yet covered in the previous round. When no remaining test case can improve the statement coverage, the technique will reset all the statements to "not covered" and reapply addtl-st on the remaining test cases. When more than one test case covers the same number of statements not yet covered, it selects one of them randomly. The additional branch test case prioritization technique (addtl-br) is the same as addtl-st, except that it uses branch coverage information instead.

For similarity-based techniques, the actual value of execution counts is used. In our techniques, we reorder program entities to obtain relative execution counts. And, as a comparison, the original execution information is adopted in the peer techniques. To investigate the impacts of different levels of granularity of coverage information, statement-level and branch-level coverage information are involved.

Whereas basic and repeated coverage maximization filter and prioritize test cases based on their incremental contribution to maximizing code coverage, similarity-based techniques select and prioritize test cases based on how their profiles are distributed in the

Table 2 Prioritization techniques

Ref.	Technique	Coverage information	Similarity measure	Algorithm
T1	addtl-st	Original statement		Additional greedy
T2	addtl-br	Original branch		Additional greedy
T3	orig-st-pb-ART	Original statement	PB distance	ART
T4	orig-br-pb-ART	Original branch	PB distance	ART
T5	orig-st-pb-ICP	Original statement	PB distance	ICP
T6	orig-br-pb-ICP	Original branch	PB distance	ICP
T7	re-st-ed-FOS	Relative statement	edit distance	FOS
T8	re-br-ed-FOS	Relative branch	edit distance	FOS
T9	re-st-ed-GOS	Relative statement	edit distance	GOS
T10	re-br-ed-GOS	Relative branch	edit distance	GOS

For abbreviations in column 2, see text.

multidimensional profile space defined by a particular dissimilarity metric. Dickinson et al. (2001) reported experiments in which different dissimilarity metrics were used for cluster filtering and failure pursuit. They found that these techniques work best overall with the proportional binary (PB) metric, which is a modified Euclidean distance formula that takes into account whether or not an entity is covered, and also the count of how many times the entity is executed while adjusting for differences in scale between counts. Another widely used metric is edit distance, which is also used in our techniques. The edit distance between two ordered sequences is the minimum number of operations required to transform one into the other; the allowable operations are insertion, deletion, and substitution (Hemmati and Briand 2010).

Note that our techniques do not adopt PB distance. After reordering the program entities, it is not suitable for our techniques to use PB distance. On the one hand, our techniques emphasize the sequence of program entities while PB distance focuses on the difference in each entity not the order of entities. On the other hand, PB measures the distance between the two corresponding position of two runs, while the corresponding relationship is disturbed in the ordered sequences of program entities.

Finally, we select two widely used algorithms, ART and ICP, for which the prioritization process is similar to the algorithms depicted in Sect. 4. The FOS technique is similar to ART-based techniques and the GOS technique is similar to ICP-based techniques.

5.3 Subject programs

In our empirical study, five Java programs are used. Three of them were downloaded from SIR (Group 2009): JTopas, XML-Security, and Ant; the others are JDepend, version 2.9.1 (Clark 2005) and Checkstyle, version 5.3 (Burn 2005). All are open source projects with self-validating test cases available. The detailed information of these five programs is listed in Table 3.

JDepend and JTopas are of low scale. JDepend traverses Java class file directories and generates design quality metrics for each Java package. JTopas is a Java library used for parsing text data. Compared with these two programs, the others are larger. XML-Security is a library implementing XML signature and encryption standards. Checkstyle is a development tool to help programmers write Java code that adheres to a certain coding

Table 3 Detailed information of subject programs

Programs	Line of codes	# Test cases	# Mutations (faults)	# Faults in loops
JDepend	2,459	47	241	90
JTopas	5,297	47	940	191
XML-Security	18,237	95	226	81
Checkstyle	43,407	164	4.873	270
Ant	75,429	101	531	206

standard. We obtained version 5.3 from its official homepage. Ant is a Java-based build tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. Version 2.9.1 of JDepend and version 5.3 of Checkstyle were downloaded from their official homepages, while JTopas, XML-Security, and Ant are available in SIR. The version of each SIR program is selected randomly: version 3 of JTopas, version 1 of XML-Security, and version 5 of Ant. Other information on these programs will be depicted in the following sections.

5.4 Test suites

Each test suite consists of JUnit tests that are obtained together with each program. However, some of them were faulty during our executions. If we discard the faulty ones straightaway, the remaining number is too small for our experiment. For example, if there is only one test left, the prioritization results of different techniques will seem meaningless.

We notice that a test case is comprised of several independent testing methods, which inspires us to decompose one original test case into several smaller ones. Each smaller test case then tests just one method. This decomposing will not lose the effectiveness of the original test suite.

For JTopas, according to the 22 original test cases, 51 are generated, of which 4 are faulty. Hence, 47 test cases remain. For JDepend, 11 original test cases are decomposed into 53 smaller ones. After discarding 6 faulty ones, the remaining number of test cases is 47. For XML-Security, 107 smaller test cases are generated and, after decomposing and discarding, the final number of test cases is 95. For Checkstyle and Ant, all the original tests are successful during our executions and these test cases are used directly in our experiment without any decomposing and discarding.

The final number of test cases for each program is shown in Table 3.

5.5 Mutant generation and versions

For the purpose of measuring the effectiveness of test case prioritization techniques in regression testing, we should use the information collected on the base version to the subsequent faulty versions. In our experiment, we use the tool Jumble (Irvine et al. 2007) to generate mutants on each subject program to form subsequent faulty versions. Jumble is a class-level mutation testing tool incorporating with JUnit. And Jumble can provide several kinds of mutations that can be used to simulate to a certain extent the modifications of a program. Andrews et al. (2006) verified that generated mutants can be used to predict the detection effectiveness of real faults, which verifies the safety of using mutations in place of real faults. Do and Rothermel (2006), focusing on the results of test case

prioritization techniques, considered whether evaluating prioritization techniques against mutation faults and seeded faults differs, and it was concluded that mutation faults can be safely used in place of real or hand-seeded faults. Based on these studies, we used mutation faults to evaluate the prioritization techniques.

One issue worth considering is the "equivalent mutant" problem. An equivalent mutant is semantically equivalent to the original program and can never be killed by any test case. Naturally, there are two approaches to handle this problem. The first is to consider mutants not killed by any test case as equivalent, while the second is to treat mutants not killed as nonequivalent. We chose the first approach in our experiment. The number of mutation faults of each subject program is listed in Table 3.

In Table 3, "Mutation faults" represent all the mutants killed by programs (not the mutants generated by Jumble) and "Faults in loops" stands for the mutants in loops. The former is used to evaluate the overall effectiveness of all test case prioritization techniques, which can help to answer RQ1 and RQ3. The latter can give insights into loops in programs and help to answer RQ2.

The number of mutants is relative to test cases and not the scale of programs. Thus, in Jumble, the number depends on the types and the number of assertions in the JUnit test cases. And the large programs (i.e., Ant) may have fewer mutant points than the small programs (i.e., Checkstyle).

Rothermel et al. (2001) and Elbaum et al. (2002) evaluated techniques for test case prioritization in two scenarios: general prioritization and version-specific prioritization. In the former scenario, the effectiveness of a test case prioritization technique is measured in terms of the rate for detecting faults over a succession of subsequent versions; in the latter scenario, the effectiveness of a technique is measured in terms of the rate for detecting faults in a particular version. In our experiment, single-fault versions, i.e., a succession of subsequent versions, are used to evaluate the effectiveness of test case prioritization techniques.

5.6 Evaluation metrics

5.6.1 Effectiveness measurement

In order to assess the fault detection rate, we use the APFD metric. As mentioned in Sect. 2, higher APFD values mean faster fault detection rates. More formally, according to Yoo and Harman (2012), let T be the test suite containing n test cases and let F be the set containing m faults revealed by T. Given an order o, let TF_i be the order of the first test case that reveals the ith fault. The APFD value of o is defined as the following equation:

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

5.6.2 Efficiency measurement

Although one technique can increase the fault detection rate and detect bugs in loops more quickly, it can be expensive to employ and may not reduce the overall cost of regression testing. In order to assess the cost-effectiveness of the proposed techniques, we use the cost model proposed in Malishevsky et al. (2002). Given an order o of test suite T, the cost of this order is defined as:

$$C^o = C_a(T) + C_p(T) + \text{delays}^o \tag{2}$$

$C_a(T)$ is the cost of analysis, including the cost of source code analysis, collection of execution traces, and other useful information collection; $C_p(T)$ is the cost of executing the prioritization algorithm. As shown in Eq. 3, the delays^o is defined as the cumulative cost of waiting for each fault to be exposed when executing T under order o.

$$\text{delays}^o = \sum_{i=1}^{m} \left(\left(\sum_{k=1}^{TF_i^o} e_k^o \right) * f_i \right) \tag{3}$$

In the equation, m is the number of faults, e_k^o is the runtime and the validation time of test case k in test suite T under order o, TF_i^o is the test number under order o which first detects fault i, and f_i is the cost of waiting a time unit for a fault i to be exposed (e.g., salaries for programmers to wait for a fault to be revealed in order to correct it).

When $\forall\, i\, f_i = 1$, delay^o sums, for each fault, the time between the start of the test suite execution and the time when this fault is first revealed. In our experiment, we suppose that the average of the run, validation, and debugging time of each test case (i.e., the value of e_k^o) is one time unit. This time unit can be an hour or other lengths of time according to the actual situation in practice. We suppose that the value of f_i is 1 in our experiment. Then, delay^o is the sum of the number of test cases before a fault can be detected.

5.7 Data collection

We use the tool CodeCover (Burn 2005) to help us instrument the source code and collect the coverage information of SUT. CodeCover is a glass box testing tool to measure the coverage information. The whole procedure consists of three main phases: instrumentation, execution, and reporting. The process starts in the instrumentation phase, which instruments auxiliary information to the source code. Six levels of granularity can be supported for the instrumentation, including statement, branch, loop coverage, term coverage (subsumes MC/DC), question mark operator coverage, and synchronized coverage. After that, executable code can be generated by a compiler. Then, the instrumented code can be executed and the coverage measurements will be recorded into a coverage log. The coverage log holds the information about the number of executions of a specific kind of entities. Finally, coverage metrics can be calculated for report generation.

5.8 Experiment setup

Our experiment is conducted using the following steps: (1) instrument each program, using the tool CodeCover mentioned above; (2) run test suites of each program against the instrumented version; (3) collect coverage information, including statement and branch coverage; (4) run test suites of each program under the Jumble framework; (5) collect mutation information and generate a fault matrix by comparing the outputs between instrumented version and each faulty version; (6) prioritize test cases by the techniques depicted in Table 2; and (7) perform analyses on the experimental results.

We carefully check the implementation of these test case prioritization techniques to guarantee the correctness of them. For the farthest-first and the ART techniques, a tie situation should be considered. For example, two candidate test cases could have the same distance from the already selected test suite. In this case, we choose one randomly. In addition, the construction of a candidate set for greedy techniques, ART techniques, and

AHC techniques involves randomness. Both of these may impact the experimental results. Hence, we run 1,000 replications of each technique to eliminate the impact of randomness.

We carried out the experiment on a MacBook Pro, running Mac OS X and equipped with an Intel processor (2.8 GHz, 2 cores) with 8 GB physical memory.

6 Result analysis

6.1 Effectiveness on fault detection rate

Figure 1 shows the box-whisker plots of the five subject programs. For each box-whisker plot, the horizontal axis represents test case prioritization techniques and the vertical axis represents APFD values. In this figure, each box-whisker plot shows the values of all the 1,000 groups of the data. From all the figures, we can find out that the farthest-first and greed-aided-clustering techniques are very stable in their rates of fault detection. However, the discrepancy within each pair is different figure by figure.

For JDepend, all the ART-based techniques can gain high APFD values and outperform ICP-based techniques. In addition, our GOS techniques based on the statement coverage can outperform other ICP techniques. Furthermore, different levels of the granularity of coverage information show few impacts on the techniques, and statement coverage even performs better than branch coverage.

For JTopas, our FOS and GOS techniques also belong to the top techniques. For ICP techniques, statement coverage can outperform branch coverage and our ICP ones can perform better than most of the others. For ART techniques and additional techniques, the granularity of coverage information shows few impacts on effectiveness.

For XML-Security, our FOS technique incorporating statement coverage and our GOS technique incorporating branch coverage show faster fault detection rate than other techniques except for the additional greedy one relying on branch coverage. And our two other techniques show low fault detection rates.

For Checkstyle, our techniques show higher effectiveness on fault detection than other similarity-based techniques, of which the FOS ones outperform the ICP-based ones. For additional greedy techniques and our FOS techniques, branch coverage can perform better than statement coverage, while other techniques have been very slightly affected by the granularity of coverage information. The additional greedy techniques perform best on fault detection rate.

For Ant, our GOS techniques show the highest fault detection rate. Our FOS techniques perform better than these two additional greedy ones, though they show low effectiveness. Although there are many overlaps between statement coverage-based techniques and branch coverage-based techniques, statement coverage shows a small improvement on branch coverage.

As observed in Fig. 1, the techniques based on ART or ICP, relying on the actual counts of program entities, show large variability compared to others, while the GOS and FOS techniques, relying on the relative execution counts of program entities, are very stable. On the one hand, the relative execution counts can characterize the program better, such that two test cases are considered as the same one by actual counts and are different by relative execution counts. On the other hand, our improved algorithm can order test cases in such a way that randomness is reduced.

In addition, techniques behave differently with different programs, which can be affected by many other factors. For example, one factor might be the percentage of test

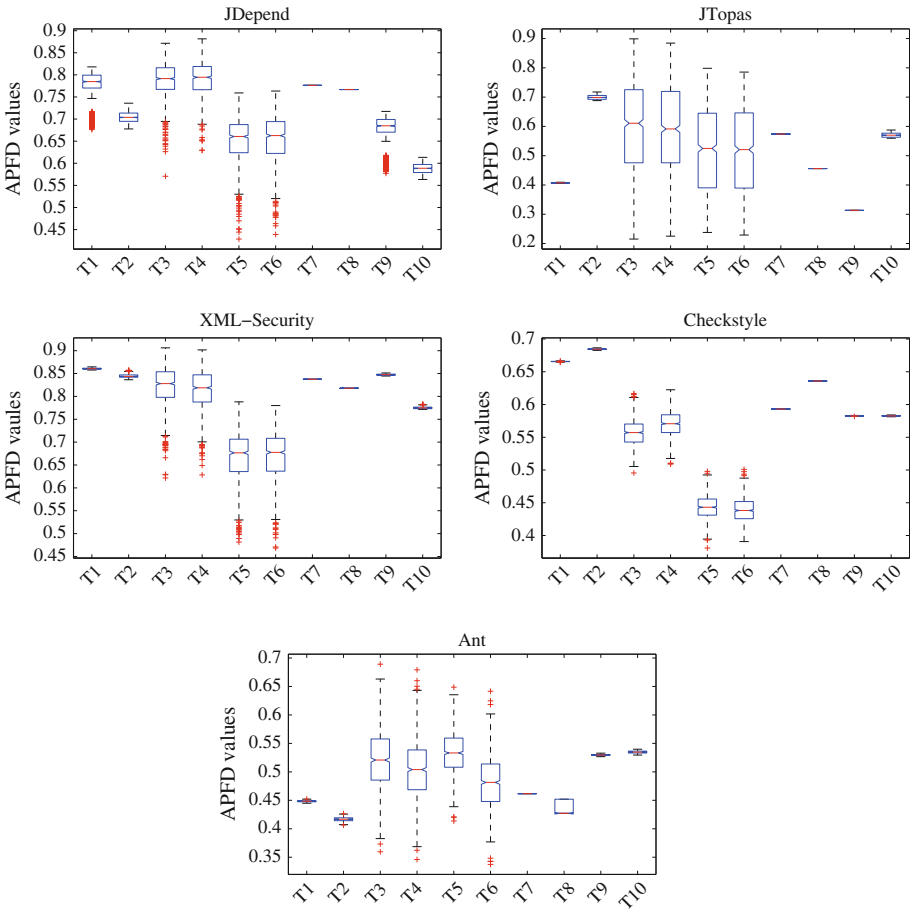

Fig. 1 APFD distributions

cases that can kill mutation faults. For example, 30 % test cases can kill mutation faults in XML-Security, while less than 30 % cannot detect mutation faults in JDepend. However, our techniques, *T*7 and *T*8, show high effectiveness under these two conditions. For JTopas, most test cases can detect mutation faults, of which only a few can detect faults in loops, such that our techniques capturing faults in loops assign low priorities to other test cases that can kill more faults and show low effectiveness. In addition, Ant is so large in size that low coverage is achieved, in which different test cases cover different codes and test cases with higher coverage kill more mutation faults. These attributes can lead the two greedy algorithms to outperform the others. Some other factors that affect these results will be discussed later.

Overall, our FOS techniques show higher fault detection than others for most of the subject programs and our GOS techniques can outperform other ICP ones. However, there are many overlaps in APFD values. To solve this problem and confirm our observations from the box-whisker plots above, we evaluated the statistical significance of our results using ANOVA analysis. The following statistical hypothesis is formulated:

H_1: APFD means for various techniques are different.

We then formulated the appropriate null hypotheses:

H_0: APFD means for various techniques are equal.

Our results indicate that null hypothesis H_0 can be rejected at the 1 % level ($p < 0.01$) for all subject programs, confirming the box-whisker-plot observations, that is, some techniques can gain statistically different rates of fault detection. With the rejection of the null hypothesis showing that some techniques produce statistically different APFD means, we still need to run a multiple-comparison procedure to determine which techniques differ from each other. Based on the conservatism and generality of commonly used means separation tests, we adopted the Bonferroni method.

We calculated the minimum statistically significant difference between APFD means through Bonferroni for each program, and the results are shown in Table 4, where the columns represent test case prioritization techniques, and the rows represent APFD values. The techniques are partitioned by APFD values, and techniques that overlap with each other are not significantly different where fault detection effectiveness varies for different techniques.

Turning to the statistical analysis in Table 4, we can confirm our observations and conclude that our FOS techniques based on the ordered sequences of program entities, especially statement entities, may have only minor actual differences from the most effective group and outperform most of the other techniques. In addition, our GOS techniques can statistically outperform other ICP techniques. And this indicates that ordered sequences of program entities measured by execution frequencies can expose additional useful information for fault detection. In addition, we observe that the granularity of coverage information has few impacts on fault detection effectiveness.

6.2 Effectiveness on detecting faults in loops

In this subsection, we investigate the capability of detecting bugs in loops for the three techniques. The reason why we chose bugs in loops for deep investigation is that these bugs are more likely to be relevant to ordered sequences of program entities measured by execution frequencies, as mentioned in Sect. 3.

Figure 2 depicts the tendency of detecting bugs in loops by each technique for each program. The horizontal axis stands for the percentage of test cases executed, and the vertical axis stands for the percentage of bugs detected in loops. Each point in this figure stands for the mean value by averaging 1,000 groups of the data.

For JDepend, XML-Security, and Checkstyle, faults are detected quickly at the early stage and quite slowly by the last test cases. This means that all the techniques can improve the fault detection rate in loops. For additional greedy techniques, the reason might be that some test cases that can detect more bugs have large additional coverage. For similarity-based techniques, these test cases with larger additional coverage might also be far from test cases with large coverages, such that they are selected earlier by these techniques.

Although various techniques can gain different rates of detecting faults in loops, they share the same tendency that all curves are below or above the diagonal, and that faults are detected quickly or slowly under a given percentage of test cases. This observation confirms that both statement coverage and branch coverage reflect the same attributes of programs, such as program structure. In addition, different algorithms can hold such

Table 4 Bonferroni means separation tests

Grouping	Means	Techniques
JDepend		
A	0.789	*T*4
A	0.7883	*T*3
B	0.7762	*T*7
B	0.7761	*T*1
C	0.7668	*T*8
D	0.7043	*T*2
E	0.6747	*T*9
F	0.6538	*T*6
F	0.6516	*T*5
G	0.5886	*T*10
JTopas		
A	0.6999	*T*2
B	0.5996	*T*3
B	0.5914	*T*4
C	0.5738	*T*7
C	0.5709	*T*10
D	0.5188	*T*5
D	0.5152	*T*6
E	0.456	*T*8
F	0.4072	*T*1
G	0.3132	*T*9
XML-Security		
A	0.8607	*T*1
B	0.8475	*T*9
B	0.8445	*T*2
C	0.8374	*T*7
D	0.8227	*T*3
E	0.8178	*T*8
E	0.8139	*T*4
F	0.7753	*T*10
G	0.6701	*T*6
G	0.6682	*T*5
Checkstyle		
A	0.6845	*T*2
B	0.6657	*T*1
C	0.6358	*T*8
D	0.5931	*T*7
E	0.5827	*T*10
E	0.5826	*T*9
F	0.57	*T*4
G	0.5569	*T*3
H	0.4428	*T*5

Table 4 continued

Grouping	Means	Techniques
H	0.439	*T*6
Ant		
A	0.5347	*T*10
A,B	0.5328	*T*5
B	0.5297	*T*9
C	0.52	*T*3
D	0.504	*T*4
E	0.4815	*T*6
F	0.4617	*T*7
G	0.4484	*T*1
H	0.4391	*T*8
I	0.4167	*T*2

properties of the programs, although they might gain different results on software tasks, such as test case prioritization.

When ordered sequences of program entities are adopted, the loop information is of high rank, while the other information holds still, so that such sequences of program entities might gain improvements in detecting faults in loops.

For JDepend, our FOS technique relying on branch coverage can detect bugs in loops much more quickly than most of other techniques, while our three other techniques can only detect faults as quickly as a few other effective techniques. For JTopas, the FOS technique utilizing statement coverage and GOS technique utilizing branch coverage can find a few bugs when only a few test cases are executed and can detect faults quite quickly as test cases are added. And these two techniques outperform others in revealing faults in loops. Our other two techniques can effectively detect faults. For XML-Security and Checkstyle, the FOS technique utilizing statement coverage can still reveal faults the most quickly. The GOS technique utilizing branch coverage performs better than most other techniques. For Ant, our techniques perform no better or even worse than other techniques.

For Ant, each JUnit test case can only test one class, that is, no two test cases test the same class. Faults in loops are evenly spread across the classes, such that the orders of test cases have few impacts on the fault detection rate and most curves are similar to the diagonal. However, the situation is different in JTopas, in which some tests can detect most of the faults in bugs. Our techniques can select these tests earlier than others, which is an advance in our techniques, while the curves for other techniques are close to the diagonal.

In conclusion, our techniques based on the ordered sequence of program entities, especially ones that adopt FOS utilizing statements and GOS utilizing branches, can find more bugs in loops than the other techniques when the same number of test cases are executed. Therefore, if regression testing is terminated because of limited resources, this FOS technique can be more effective in detecting bugs in loops than the peer techniques under the same conditions.

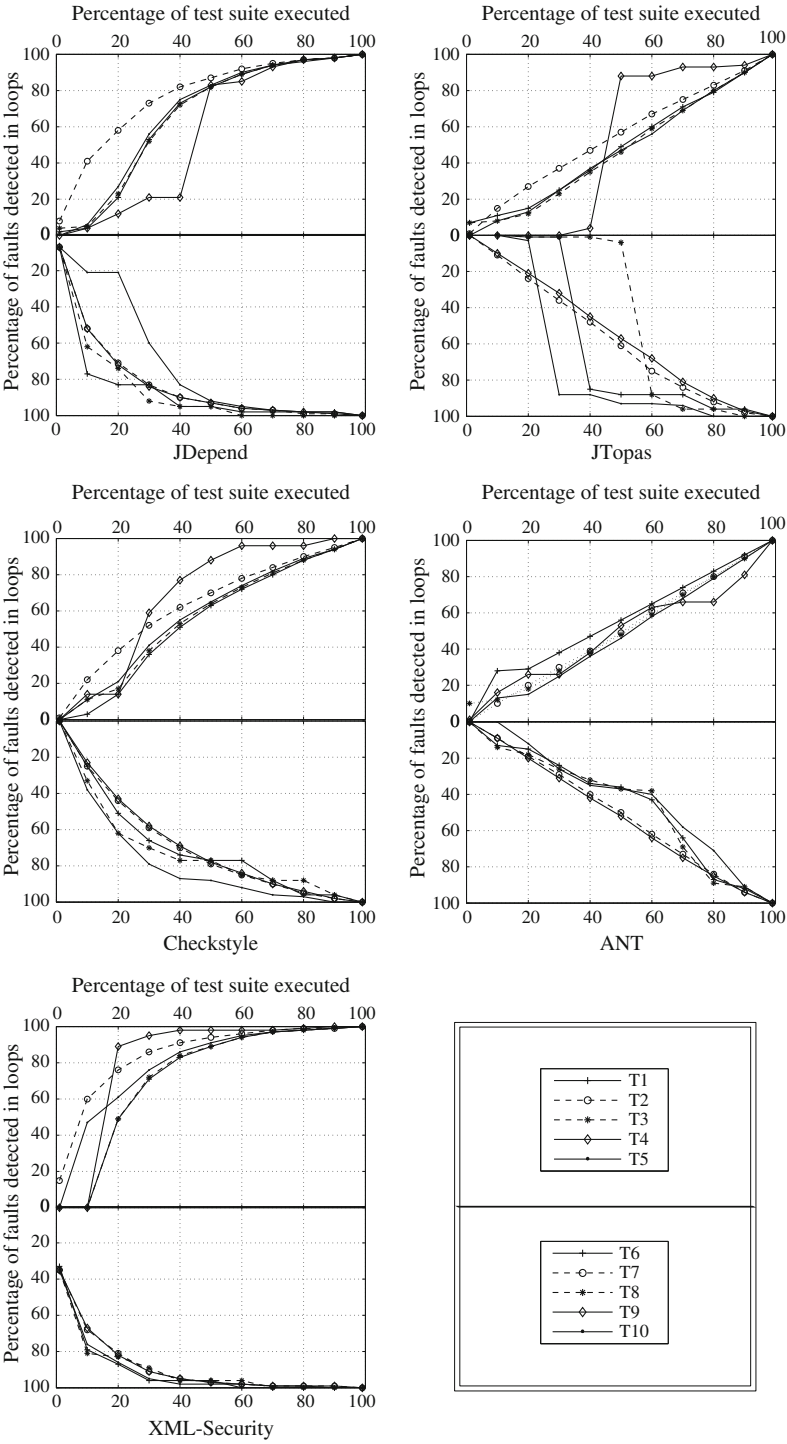

Fig. 2 Detecting faults in loops

6.3 Factors affecting effectiveness

In this section, we investigate the impacts on the effectiveness of test case prioritization techniques based on the factors of granularity of coverage information and algorithm.

To compare the impacts of granularity of coverage information on APFD results of test case prioritization techniques, techniques that only have different levels of granularity of coverage information are compared and the results are shown in Table 5. If the mean APFD value of statement coverage is higher (lower, respectively) than that of branch coverage at 5 % significance level, a "$>$" ("$<$") sign is put in the corresponding cell. In cases in which there is no significant difference between their mean values, an "=" sign is put in the cell.

As shown in Table 5, there are 6 "$<$" signs, 11 "$>$" signs, and 8 "=" signs. For XML-Security and Ant, statement coverage outperforms branch coverage; for Checkstyle, branch coverage performs better than statement coverage; for the other two programs, it is hard to tell which level of granularity of coverage information is better. In addition, for each type of techniques, it also vague whether one level of granularity can outperform another. Furthermore, according to Fig. 1 and Table 4, the differences between various levels of coverage information are quite small, although some are statistically different. In conclusion, different levels of coverage information have only a few impacts on the effectiveness of test case prioritization techniques. Although branch-level techniques are theoretically comparable to statement-level techniques, the latter is slightly preferable and performs better in our experiment.

Notice that FOS cannot be compared with other techniques for their different coverage information. To facilitate the next study, the FOS technique is considered as a kind of ART-based technique and the GOS technique is considered as a kind of ICP-based technique.

To compare the impacts of algorithms on APFD results of prioritization techniques, techniques that only have different algorithms are compared (additional greedy techniques are omitted for the same reason above). There are only two algorithms to evaluate: ART and ICP. ART and FOS are denoted by ART, and GOS and ICP are denoted by ICP. Similar to Table 5, we put "$<$", "$>$" and "=" in the cell, and the results are shown in Table 6.

As shown in Table 6, there are 4 "$<$" signs, 16 "$>$" signs, and 0 "=" signs. Obviously, most ART techniques can outperform ICP ones. In other words, with the same coverage information and distance metric, the effectiveness of various techniques is dominated by algorithms, and ART techniques show significant improvement in the results. With our improvement in ICP, ICP techniques can outperform ART techniques in some situations, such as the Ant program.

Table 5 Comparison between statement and branch (st:br)

Ref.	JDepend	JTopas	XML-Security	Checkstyle	Ant
addtl	$<$	$<$	$>$	$<$	$>$
orig-pb-ART	=	=	$>$	$<$	$>$
orig-pb-ICP	=	=	=	=	$>$
re-ed-FOS	$>$	$>$	$>$	$<$	$>$
re-ed-GOS	$>$	$<$	$>$	=	=

See text for explanation of symbopls and abbreviations

Table 6 Comparison between algorithms (ART:ICP)

Ref.	JDepend	JTopas	XML-Security	Checkstyle	Ant
orig-pb-st	>	>	>	>	>
orig-pb-br	>	>	>	>	>
re-ed-st	>	>	<	>	<
re-ed-br	>	<	>	>	<

See text for explanation of symbopls and abbreviations

To conclude this section, algorithms do have significant impacts on the effectiveness of test case prioritization techniques, while the granularity of coverage information shows few impacts. As practical guides, the ART algorithm incorporating ordered sequences of program entities is preferable for similarity-based techniques, and to facilitate the collection of coverage information, the statement coverage is preferable.

6.4 Efficiency

Although our techniques can increase the rate of fault detection and effectively detect bugs in loops, it may be expensive to employ and may not reduce the overall regression testing cost. We use the model depicted in Sect. 5.6.2 to assess the cost-effectiveness of the proposed techniques. The units of $C_a(T)$ and C_p are both *ms*, and the delays^O is depicted in Sect. 5.6.2. The results are shown in Table 7.

C_a is the cost for generating a distance matrix according to a given distance metric in our experiment. As shown in Table 7, generating a distance matrix for statement coverage needs much more time than that for branch coverage. In addition, our techniques need no extra cost compared to other similarity-based techniques. For programs of small size, the cost is extremely smalle, 2 ms and 4 ms; for the program with tens of thousands of lines, the cost is less than 20 s. And the cost for generating a matrix for similarity-based techniques is acceptable.

C_p is the cost of running each technique 1,000 times. On the one hand, running some techniques just once is difficult to record. On the other hand, running 1,000 times can amplify the differences between different techniques. According to Table 7, it is noticeable that some ART-based techniques are faster than the greedy techniques. We will explain this later. The ICP-based techniques seem to need a lot of time. However, the most complex technique only needs less than 11 s for one run. And it involves no extra cost for our GOS techniques by introducing additional greedy algorithm into ICP. Also, our FOS techniques show the same efficiency as other techniques.

However, C_a and C_p will be amortized over the subsequent releases during the whole regression testing process (Elbaum et al. 2002). Consequently, delays^o is a key factor that reflects the efficiency of the techniques. The lower the delays^o values, the higher the efficiency of the corresponding techniques. As it is observed that algorithms have significant impacts on the effectiveness of test case prioritization techniques, techniques adopting the same algorithms share similar efficiencies. For most of the subject programs, ART-based techniques are more efficient than ICP-based techniques, and our techniques can perform as efficiently as other similarity-based techniques. In other words, our techniques can be cost-effective.

Table 7 The cost of prioritization techniques for each program

Ref.	JDepend			JTopas			XML-Security			Checkstyle			Ant		
	C_a	C_p	delays^O	C_a	C_p	delays^O	C_a	C_p	delays^O	C_a	C_p	delays^O	C_a	C_p	delays^O
*T*1	0	1,190	3,766	0	3,781	387	0	20,911	11,993	0	176,913	291,113	0	10,2746	29,706
*T*2	0	467	3,639	0	1,830	411	0	6,565	11,613	0	72,031	274,182	0	35,676	30,066
*T*3	6	82	3,087	22	202	246	113	528	17,417	862	2,239	384,646	562	2201	26,359
*T*4	2	44	3,013	10	106	326	34	207	17,189	372	1,132	381,815	183	764	27,216
*T*5	6	7,267	3,810	22	7,556	446	113	79,048	23,938	849	103,741,3	398,286	567	136,703	27,836
*T*6	2	8,288	3,802	9	8,437	446	35	81,087	23,628	363	106,572,7	401,840	190	152,423	27,273
*T*7	37	94	3,081	260	182	266	5,051	527	16,718	15,775	2,317	390,070	13,526	2112	27,178
*T*8	2	48	3,078	34	102	327	254	198	16,289	489	1,022	386,307	944	760	27,813
*T*9	39	7,500	3,861	282	7,675	228	5011	86,902	21,628	15,867	740,365	387,752	13,236	156,436	23,000
*T*10	2	8,701	4,080	34	8,305	416	258	90,910	20,604	485	100,004,7	370,590	946	157,984	22,082

As the size of program increases, C_p of ICP-based techniques grows faster than others. Another phenomenon is that some ART techniques can cost less than additional greedy techniques. Notice that the cost of ART-based techniques is related to the size of the distance matrix, which is determined by the number of test cases. The cost of additional greedy techniques is determined by both the number of test cases and the number of statements (branches). Whereas their complexities are both $O(n^2)$, the cost of ART-based techniques is less than additional greedy ones. In addition, C_p of Checkstyle is more than that of Ant, although Checkstyle is of larger size. This means that the size of the test suite has many more impacts on the cost of prioritization techniques than the scale of programs.

In conclusion, our techniques can perform as efficiently as the corresponding techniques (i.e., st-ART-based techniques, *T*3 and *T*7). In addition, our techniques outperform most other techniques in the previous sections. In other words, we can conclude that FOS techniques can be more cost-effective than the peer techniques, which means that we can reduce the overall testing cost by using our techniques based on the improved algorithms incorporating ordered sequences of program entities.

6.5 Practical guides

Based on all the conclusions above, we can obtain the following practical guidelines for selecting suitable test case prioritization techniques for different applications:

- More attention should be paid to algorithms other than the granularity of coverage information. FOS can outperform others, statement coverage is more preferable as is cost-effectiveness, and our ordered sequences measured by edit distance are preferred.
- To effectively detect bugs in loops, our FOS technique is a good choice. Moreover, our FOS technique can work better than the existing techniques for other faults.
- To reduce the cost of test case prioritization techniques, additional greedy techniques should be considered, especially when test suite size is large. FOS can be selected for its cost-effectiveness.

Generally speaking, our FOS incorporating ordered sequences of statements can work effectively and efficiently in practice.

6.6 Threats to validity

One threat to internal validity is the correctness of the implementations of our techniques, which may be accompanied by internal faults. To minimize the threat, we carefully inspected our programs and results in a team meeting.

One threat to external validity is the representativeness of the subject programs and test suites. We used five Java programs as subject programs and used the existing test cases along with each program without any augmentation. Specifically, for two programs, we decomposed test cases into smaller ones to ease implementation. The sizes and types of programs and test cases may affect the experimental results and we will study this in the future.

One threat to construct validity is concerned with the evaluation measurements of our experiments. To minimize the threats, we introduced a widely used metric, APFD, to evaluate the effectiveness of test case prioritization techniques. A comprehensive cost metric was also used to measure the efficiency of test case prioritization. In the future, we might use other cost–benefit models to assess the cost-effectiveness of our techniques. Another threat to construct validity might involve regression faults. In our experiment, for

the general purpose of test case prioritization, we select only one base version of each subject program, and generated a series of subsequent versions by mutation testing based on the single base version. Although this can simulate regression faults, multiple versions with real faults can involve regression faults. This is left for our future work.

7 Conclusion and future work

In this paper, we proposed several novel similarity-based test case prioritization techniques based on the ordered sequences of program entities. The execution profile of each test case is first transformed into ordered sequences of program entities, sorted by the execution counts of each entity. Then, edit distance is used to calculate pair-wise distances. Two algorithms, FOS and GOS, are proposed to fulfill the test case prioritization techniques. Furthermore, we conducted an empirical study and the results showed that our new techniques can increase the fault detection rate more significantly and detect bugs in loops more quickly than the existing techniques. At the same time, our new techniques can be more cost-effective than the peer ones. Finally, some practical guides are presented. Generally speaking, FOS incorporating ordered sequences of statements is preferable.

Whereas our results are positive, it can be a possible future research direction to work on multiple versions and regression faults to further investigate these test case prioritization techniques in practice. In addition, how these factors, the number of mutation faults and the number of test cases, affect the results of test case prioritization techniques will be investigated.

Acknowledgments The authors would like to thank the anonymous reviewers for their constructive comments. The work described in this article was partially supported by the National Basic Research Program of China (973 Program 2014CB340702), the National Natural Science Foundation of China (61170067, 61373013) and the Scientific Research Foundation of Graduate School of Nanjing University (2013CL13).

References

Andrews, J., Briand, L., Labiche, Y., & Namin, A. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering, 32*(8), 608–624.

Burn, O. (2005). *Checkstyle homepage*. http://checkstyle.sourceforge.net/. Accessed Nov 2011.

Carlson, R., Do, H., & Denton, A. (2011). A clustering approach to improving test case prioritization: An industrial case study In *Proceedings of the 27th International Conference on Software Maintenance (ICSM'011)*, 2011, pp. 382–391. IEEE.

Chen, Z., Duan, Y., Zhao, Z., Xu, B., & QIAN, J. (2011). Using program slicing to improve the efficiency and effectiveness of cluster test selection. *International Journal of Software Engineering and Knowledge Engineering, 21*(6), 759.

Clark, M. (2005). *Jdepend homepage*. http://clarkware.com/software/JDepend.html. Accessed 23 Jan 2012.

Dickinson, W., Leon, D., & Podgurski, A. (2001). Pursuing failure: The distribution of program failures in a profile space. In *ACM SIGSOFT Software Engineering Notes*, (Vol. 26, pp. 246–255). ACM.

Do, H., Rothermel, G., & Kinneer, A. (2004). Empirical studies of test case prioritization in a junit testing environment. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, 2004, pp. 113–124. IEEE.

Do, H., & Rothermel, G. (2006). On the use of mutation faults in empirical a ssessments of test case prioritization techniques. *IEEE Transactions on Software Engineering, 32*(9), 733–752.

Do, H., Mirarab, S., Tahvildari, L., & Rothermel, G. (2010). The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering, 36*(5), 593–617.

Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering, 28*(2), 159–182.

Fang, C., Chen, Z., & Xu, B. (2012). Comparing logic coverage criteria on test case prioritization. *Science China Information Sciences, 55*(12), 2826–2840.

Group, G. R., et al. (2009). Software-artifact infrastructure repository (sir). http://sir.unl.edu.

Hemmati, H., Arcuri, A., & Briand, L. (2010). Reducing the cost of model-based testing through test case diversity. *Testing Software and Systems, 6435*, 63–78.

Hemmati, H., Briand, L. (2010). An industrial investigation of similarity measures for model-based test case selection. In *Proceedings of the 8th 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, 2010, pp. 141–150. IEEE.

Irvine, S., Pavlinic, T., Trigg, L., Cleary, J., Inglis, S., & Utting, M. (2007). Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, (TAICPART-MUTATION'07)*, 2007, pp. 169–175. IEEE.

Jiang, B., Zhang, Z., Chan, W., & Tse, T. (2009). Adaptive random test case prioritization. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09)*, 2009, pp. 233–244. IEEE.

Jones, J., & Harrold, M. (2003). Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering, 29*(3), 195–209.

Leon, D., Podgurski, A. (2003). A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, 2003, pp. 442–453. IEEE.

Li, Z., Harman, M., & Hierons, R. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering, 33*(4), 225–237.

Lo, D., Cheng, H., Han, J., Khoo, S., Sun, C. (2009). Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th International Conference on Knowledge Discovery and Data Mining (SEKE'09)*, ACM, 2009, pp. 557–566.

Malishevsky, A., Rothermel, G., Elbaum, S. (2002). Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, 2002, pp. 204–213. IEEE.

Marick, B. (1994). *The craft of software testing: Subsystem testing including object-based and object-oriented testing*. Champaign: Prentice-Hall.

Masri, W., Podgurski, A., & Leon, D. (2007). An empirical study of test case filtering techniques based on exercising information flows. *IEEE Transactions on Software Engineering, 33*(7), 454–477.

Renieres, M., Reiss, S. (2003) Fault localization with nearest neighbor queries. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE'03)*, 2003, pp. 30–39. IEEE.

Rothermel, G., Untch, R., Chu, C., & Harrold, M. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering, 27*(10), 929–948.

Sampath, S., Bryce, R., C., Viswanath, G., Kandimalla, V., & Koru, A., G., (2008). Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST'08)*, 2008., pp. 141–150. IEEE.

Sumner, W., Bao, T., Zhang, X. (2011). Selecting peers for execution comparison. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11)*, 2011, pp. 309–319. IEEE.

Wagner, R., & Fischer, M. (1974). The string-to-string correction problem. *Journal of the ACM, 21*(1), 168–173.

Wu, K., Fang, C., Chen, Z., Zhao, Z. (2012) Test case prioritization incorporating ordered sequence of program elements. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST'12)*, 2012, pp. 124–130.

Yan, S., Chen, Z., Zhao, Z., Zhang, C., Zhou, Y. (2010). A dynamic test cluster sampling strategy by leveraging execution spectra information. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*, 2010, pp. 147–154. IEEE.

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability, 22*(2), 67–120.

Yoo, S., Harman, M., Tonella, P., Susi, A. (2009). Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the 8th International Symposium on Software Testing and Analysis (ISSTA'09)*, 2009, pp. 201–212. ACM.

Zhang, C., Chen, Z., Zhao, Z., Yan, S., Zhang, J., Xu, B. (2010). An improved regression test selection technique by clustering execution profiles. In *Proceedings of the 10th International Conference on Quality Software (QSIC'10)*, 2010, pp. 171–179. IEEE.

Author Biographies

Chunrong Fang is currently a Ph.D. student in Software Engineering at Nanjing University. He received a B.S. degree in Software Engineering from Nanjing University in 2010. His research is focused on software testing, especially on test case evolving and test criteria.

Zhenyu Chen is currently an Associate Professor at the Software Institute, Nanjing University. He received his B.Eng. and Ph.D. in Mathematics from Nanjing University. He worked as a Postdoctoral Researcher at the School of Computer Science and Engineering, Southeast University, China. His research interests focus on software analysis and testing. He has more than 60 publications at major venues, such as ACM TOSEM. He was the PC co-chairs of QSIC 2013, AST 2013, and IWPD 2012. He has also served on the program committee of many international conferences. Professor Chen has won research funding from several competitive sources such as NSFC.

Kun Wu currently works at Microsoft, Windows Fundamental team and is in charge of optimizing Windows-related products. She received her master's degree of Software Engineering from Nanjing University in June 2012. She has received several awards at Nanjing University.

Zhihong Zhao is currently a Professor at the Software Institute, Nanjing University. He received his B.Eng. and Ph.D. in the Department of Computer Science from Nanjing University. His research interests include software engineering and data mining. He has more than 40 publications at major venues, such as ICST, QSIC, SEKE, etc. Professor Zhao has won research funding from several competitive sources such as NSFC.