Adaptive Random Test Case Prioritization[*]

Bo Jiang, Zhenyu Zhang
The University of Hong Kong
Pokfulam, Hong Kong
{bjiang, zyzhang}@cs.hku.hk

W. K. Chan[†]
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Abstract—Regression testing assures changed programs against unintended amendments. Rearranging the execution order of test cases is a key idea to improve their effectiveness. Paradoxically, many test case prioritization techniques resolve tie cases using the random selection approach, and yet random ordering of test cases has been considered as ineffective. Existing unit testing research unveils that adaptive random testing (ART) is a promising candidate that may replace random testing (RT). In this paper, we not only propose a new family of coverage-based ART techniques, but also show empirically that they are statistically superior to the RT-based technique in detecting faults. Furthermore, one of the ART prioritization techniques is consistently comparable to some of the best coverage-based prioritization techniques (namely, the "additional" techniques) and yet involves much less time cost.

Keywords—*Adaptive random testing; test case prioritization*

I. INTRODUCTION

Regression testing is an important and yet time-consuming software development activity [21][23]. It executes an existing test suite (denoted by T) on a changed program (denoted by P) to assure that the program is not adversely affected by unintended amendments. For instance, the retest-all strategy executes all available test cases in T [15][20][21]. Test suites can be large and conducting regression tests is tedious. To address this problem, existing research studies consider different dimensions to make regression testing more viable to software development. Techniques may execute a subset of T on P (regression test selection [23]), remove some test cases from T permanently (test suite reduction [14]), assign the execution priority of the test cases in T (test case prioritization [13]), or use a combination of these.

Test case selection and reduction may not execute P over certain test cases of T. Although either strategy can make regression testing faster to complete, the fault detection ability of T is generally compromised. Test case prioritization reorders T for execution to maximize a chosen testing goal (denoted by G) [11][13][28]. G can be technical (e.g., maximizing the code coverage rate on a given version of the

software) or business-oriented (e.g., minimizing early human participation in the testing phase). Test case prioritization does not discard any test case, and hence the fault detection ability of T is not compromised.

To present our work clearly, let us first define a few terms. Suppose $T = \{t_1, t_2, \ldots, t_n\}$ is a regression test suite with n test cases. A test sequence S is an ordered set of test cases. If t is a test case, and $S = \langle s_1, s_2, \ldots, s_k \rangle$, we define $S\char`^t$ to be $\langle s_1, s_2, \ldots, s_k, t \rangle$. Furthermore, we use the notation T/S to represent the maximal subset of T whose elements are not in S. A goal evaluation function g is a function that accepts a test goal G and a test sequence S, and returns a natural number N which represents how well S scores with respect to G. Without loss of generality, we further assume the larger such a number, the better S satisfies G.

When discussing test case prioritization techniques, we distinguish two cases, namely, general prioritization and version specific prioritization [7]. The former aims at selecting a test case ordering that will be effective (on average) over a succession of subsequent versions of the software. It is particularly applicable when the code bases of subsequent versions are unavailable at the time of test case prioritization. The later is concerned with a particular version of the software for which we may wish to prioritize test cases in a manner that will be most effective.

In this paper, we study general test case prioritization because it is useful irrespective of the availability of change information. For instance, a developer (or a software vendor) may not release the source code of a changed version to another developer (or the client) until the latter agrees to accept the functionality of the changed version. In such a scenario, we may conduct general test case prioritization on the original version to speed up the process to see whether there are faults that remain in the changed version.

Greedy algorithms [17] are a class of coverage-based test case prioritization techniques that have been widely studied in the public literature. Examples include the total-statement coverage technique and the additional-statement coverage technique [12]. Suppose T is the given regression test suite and a test sequence S has been selected using a technique in this class. Such a technique picks t' from T/S as the next test case to add to S if $S\char`^t'$ yields the maximum value in evaluation goal. In other words, the technique selects $t' \in T/S$ if $g(G, S\char`^t') \geq g(G, S\char`^x)$ for all $x \in T/S$.

We observe that in the class of greedy algorithms, *multiple* test cases in T/S (say t' and t'') may result in the same maximum value in evaluation goal (that is, $g(G, S\char`^t')$

= $g(G,\ S\hat{\ }t'')$). In such a tie case, to the best of our knowledge, almost all existing techniques [17] randomly pick one test case (say t'') to resolve the tie (that is, to construct $S\hat{\ }t''$). The additional-statement prioritization technique [11] further reset the coverage information of the remaining test cases when none of them improves the coverage of the test cases already selected. Although g is no longer a monotonic function, for each round of selection of new test cases, g could still be used as if it were monotonic. In any case, the "universal" adoption of random selection to resolve tie-cases remains unchanged.

Existing research shows empirically that random test case prioritization (*a.k.a.* random ordering) can be *ineffective*. It has been a long tradition to deem random ordering as the lower bound control technique [9][10][11][12][13][17][22][24]. If random ordering is indeed ineffective, we would like to ask the question: *Why are other techniques not used to resolve tie cases?*

Chen et al. [2][3] propose the concept of Adaptive Random Testing (ART) to replace random testing for test case generation. The basic idea of ART is to spread the distribution of test cases as evenly as possible across the *input domain*. ART improves over random testing by 40 to 50% in terms of the number of test cases to reveal the first failure (the F-measure) [3]. Chen and Merkel [5] further prove mathematically that no technique can be better than random by more than 50% of the F-measure. Thus, the improvement by ART over random is close to the theoretical limit..

If ART can improve random testing in exposing faults faster for test case generation, why do we not adapt it for test case prioritization? Previous ART techniques for test case generation use (black-box) input information to guide the test case selection process, and yet the input spaces can take on different forms (e.g., structured texts, video streams, images, and objects). Although distance metrics that measure test case distances in the input domain can be developed, the design complexity to handle such rich kinds of inputs and compute such distances for real-life programs may be high. On the other hand, white-box coverage information of the previously executed test cases (e.g., which statements have been exercised by which test cases) can be readily profiled, and many existing test case prioritization techniques use coverage information to achieve good prioritization results [11][13]. We thus ask a follow-up question: *Why is such low-cost white-box information not used to spread test cases across the code coverage space instead of the input space of the program?*

In this paper, we propose a set of ART prioritization techniques guided by white-box coverage information. We also conduct an empirical study to evaluate their effectiveness. Rather than integrating with techniques with the class of greedy algorithms, we choose to study them in a stand-alone fashion so the observations drawn from the study will be independent of the latter techniques.

The main contribution of this paper is twofold: (i) It proposes the first set of coverage-based ART techniques for test case prioritization. (ii) It reports the first empirical study on ART-based prioritization techniques. The results show that our techniques are superior to random ordering in terms of earlier detection of failures. One of the studied ART prioritization techniques is statistically comparable to the best-studied coverage-based prioritization techniques (namely, the additional techniques) in terms of the fault detection rate, and is much more efficient.

We organize the rest of paper as follows: Section 2 presents a motivating study to show how white-box ART can be effective in test case prioritization. Section 3 describes our white-box ART test case prioritization techniques. Section 4 presents our empirical study as well as a results analysis. Section 5 describes related work, followed by the conclusion in Section 6.

Program: Character Count	Test Cases						
main( ) {	t1	t2	t3	t4	t5	t6	t7
1 int let, dig, other, c;							
2 let = dig = other = 0;	•	•	•	•	•	•	•
3 while((c=getchar( )) != '\0'){	•	•	•	•	•	•	•
4 if('A'<=c && c<='Z')	•	•	•	•	•	•	•
5 let+=1;	•	•		•			
6 else if ('a'<=c && c<='z')	•	•		•	•	•	
7 let+=1;	•	•			•		
8 else if('0'<=c && c<='9')	•	•	•	•		•	•
9 **dig+=2;** /*Bug, should be dig+=1 */			•		•		•
10 else if(isprint(c))	•	•	•	•		•	•
11 other+=1;}					•	•	
12 printf("%d letters, %d digits, %d others\n", let, dig, other); }	•	•	•	•	•	•	•
PASS (✓)/FAIL (✗) STATUS:	✓	✓	✗	✗	✓	✓	✗

Figure 1. Motivating Example.

II.   MOTIVATING STUDY

This section shows how ART-based test case prioritization can be effective in detecting failures. Figure 1 shows a program with an off-by-one fault in statement #9. The source code is on the left and seven test cases with their statement coverage information are on the right.

A. Greedy Techniques

Total-statement (total-st) and additional-statement (addtl-st) [11][13] are two representative techniques in the class of greedy algorithms. The former prioritizes a test suite T in descending order of the number of statements covered by individual test cases. The latter prioritizes T in descending order of the number of additional statements covered by remaining test cases (relative to the test cases already selected and with a reset of the coverage information if no more coverage can be gained by any remaining test cases). Either technique resolves tie cases randomly.

For instance, using total-st, a test sequence ⟨t5, t6, t1, t2, t3, t4, t7⟩ can be generated for our example. Since the number of statements covered by the failure-causing test case t7 is less than the test cases t1, t2, t5, and t6, the greedy algorithms cannot select it earlier than the others. Similarly,

Algorithm: prioritize	Procedure: generate	Procedure: select
Inputs: U: {u_1, u_2, ...} is a set of test cases (untreated test suite)	**Inputs**: U:{u_1, u_2, ...} is a set of test cases (untreated test suite)	**Inputs**: P: {p_1, p_2, ...} is a sequence of test cases (prioritized test suite)
Output: P: ⟨p_1, p_2, ...⟩ is a sequence of test cases (prioritized test suite)	**Output**: C: {c_1, c_2, ...} is a set of test case (candidate set)	C: {c_1, c_2, ...} is a set of test case (candidate set)
		Output: s: test case (selected test case)
1. C: {c_1, c_2, ...} is a set of test cases 2. C ← **generate** (u) 3. P ← ∅ 4. s ← **select** (P, C) 5. U ← U \ {s} 6. P ← ⟨p_1, p_2, ..., $p_{\|P\|}$, s⟩ 7. **If** U ≠ ∅ **then** goto step 4 8. **Return** P	1. S: {s_1, s_2, ...} is set of statements 2. S': {s'_1, s'_2, ...} is set of statements 3. S' ← ∅ 4. randomly select u_i from U 5. set S as the statements covered by u_i 6. **If** S' ∪ S = S' **then** goto step 10 7. S' ← S' ∪ S 8. C ← C ∪ {u_i} 9. goto step 4 10. **Return** C	1. D: $d_{\|P\|,\|C\|}$ is a $\|P\| \times \|C\|$ dimensioned array 2. **For** i = 1, 2, ..., $\|P\|$ 3. **For** j = 1, 2, .., $\|C\|$ 4. $d_{i,j} ← f_1 (p_i, c_j)$ 5. k←f_2(D) 6. **Return** c_k

Figure 2. The White-Box ART Algorithm.

using addtl-st technique, one possible test sequence is ⟨$t2$, $t5$, $t4$, $t6$, $t3$, $t1$, $t7$⟩. Owing to its greedy nature, the first test case (even with reset) will always be one of $t1$, $t2$, $t5$, or $t6$.

B. ART-based Techniques

Let us consider a white-box ART test case prioritization that each time selects the next test case (from a candidate set of not yet selected test cases) that is farthest away from the already prioritized test cases. To measure the distance of two test cases, we first associate each test case with a set of statements exercised by the test case (in a previous version). We then use the Jaccard distance of the two set as the distance of two test cases. The Jaccard distance between two set (of statements) A and B is defined as $D(A, B) = 1 − |A \cap B| / |A \cup B|$. For instance, the distance between $t1$ and $t2$ is 0 because they cover exactly the same set of statements.

For illustration purpose, we further suppose that the size of the candidate set is 2, and initially the technique randomly generates a candidate set C, say, {$t1$, $t4$}. Since initially, the constructing test sequence S is empty, the technique randomly selects a test case, say $t1$, from C. Thus, S becomes ⟨$t1$⟩. The technique then randomly generates a new candidate set, say {$t4$, $t5$}.

Since the distance between $t1$ and $t5$ is $1 − 8/10$ (which is, 2/10) and the distance between $t1$ and $t4$ is $1 − 7/10$ (which is 3/10) and is larger than 2/10. The technique picks $t4$, and S becomes ⟨$t1$, $t4$⟩. Suppose that in the next round, the technique builds a new candidate set {$t6$, $t7$}, and it wants to find out which test case is farthest away from any test cases in S. There are several strategies. For instance, we may choose to maximize the minimum, average, or maximum distance between a candidate test case and any test case in S. The distance between $t1$ and $t6$ is 2/10, that between $t4$ and $t6$ is $1 − 7/10 = 3/10$, that between $t1$ and $t7$ is 3/10, and that between $t4$ and $t7$ is 0. The minimum, average, and maximum distances between $t6$ and ⟨$t1$, $t4$⟩ are 2/10, 5/20, and 3/10, respectively. The minimum, average, and maximum distances between $t7$ and ⟨$t1$, $t4$⟩ are 0, 3/20, and 3/10, respectively. To maximize the minimum distance, the technique can choose $t6$. To maximize the average distance,

the technique can also choose $t6$. To maximize the maximum distance, the technique can choose either $t6$ or $t7$.

We observe from the example, coverage-based ART techniques have two advantages. First, they cluster the test cases into several groups according to certain distance metrics. Each time, an ART-based technique prefers selecting test cases from an uncovered nonparametric partition (relative to the test cases already selected). Second, it transforms test case prioritization to a search process for a globally optimal solution. The randomness inherent to ART helps the prioritization algorithm to get out of the local maximum (often faced by the greedy algorithms). As shown in the next section, the randomness of our technique comes from the random selection of test cases to build the candidate test set while the "adaptive" is achieved by the adoption of coverage information to guide prioritization.

III. ART FOR TEST CASE PRIORITIZATION

In this section, we present our family of ART-based test case prioritization techniques.

A. White-Box ART Algorithm

The ART algorithm proposed in [3] for test case generation needs the success and failure information of the test cases already selected. In this study, since we want to present a family of ART techniques for test case prioritization in general (rather than specific to a particular amended version of a program), we avoid using the success and failure information of a test case. In our ART algorithm, therefore, we aim at selecting a test case farthest away from all prioritized test cases using coverage information.

Our algorithm, as shown in Figure 2, is summarized as follows: The main algorithm is **prioritize**, which prioritizes a given set of test cases. It iteratively builds a candidate set of test cases and, in turn, picks one test cases out of the candidate set until all given test cases have been selected. It invokes a procedure **generate** to build the candidate set. The **generate** procedure constructs a set of not-yet-selected test cases iteratively, by randomly adding remaining test cases into the candidate set as long as they can increase program coverage and the candidate set is not yet full. To decide

which candidate test case to be selected, the **prioritize** algorithm calls another procedure **select**. The latter requires a function f_1 that calculates the distance between a pair of test cases and a function f_2 that returns the index of the selected test case farthest away from the prioritized set. Function f_1 in the experiment (Section 5) uses the Jaccard distance between two test cases based on the coverage constructs used. Suppose the set of statements (or functions or branches) covered by test case p_j and c_i are $S(p_j)$ and $S(c_i)$, respectively. We have

$$f_1(p_j, c_i) = 1 - |S(p_j) \cap S(c_i)| / |S(p_j) \cup S(c_i)|$$

In Figure 1, for example, the number of statements covered by both test case $t1$ and test case $t5$ is 8, and the number of statements covered by either test case $t1$ or test case $t5$ is 10. The resulting Jaccard distance between $t1$ and $t5$ will be $1 - 8/10 = 0.2$.

Function f_2 can be defined by one of the following means, as explained in Section B(3) below.

$$f_2(D) = \begin{cases} j \ s.t. \ \min_{0 \le i \le |P|} d_{ij} = \max_{0 \le j \le |C|} \{ \min_{0 \le i \le |P|} d_{ij} \} & (1) \\ j \ s.t. \ \arg_{0 \le i \le |P|} d_{ij} = \max_{0 \le j \le |C|} \{ \arg_{0 \le i \le |P|} d_{ij} \} & (2) \\ j \ s.t. \ \max_{0 \le i \le |P|} d_{ij} = \max_{0 \le j \le |C|} \{ \max_{0 \le i \le |P|} d_{ij} \} & (3) \end{cases}$$

For a test suite having m test cases and a program having n statements, the time complexity of the algorithm is $O(m^2)$ in the best case and $O(m^3 n)$ in the worst case.

B. Features of ART Techniques

In this section, we discuss several features of the white-box ART algorithm above.

1) Size of candidate set: The size of the candidate set proposed by Chen et al. [3] is 10, which is a limit based on empirical findings. In our algorithm, we build the candidate set iteratively by randomly selecting one test case into the candidate set every time, until the candidate set is full or the selected test case cannot increase the statement, function, or branch coverage of the candidate set. In other words, unlike many existing ART techniques, ours is non-parametric with respect to the size of the candidate set. In our experiment, we use statement coverage for evaluation.

2) Test case distance f_1 : There are many ways to measure the distance between two test cases. Finding suitable similarity metrics is a research topic by itself. In our experiment, we use the Jaccard distance of the two sets of coverage information to measure the distance between two test cases.

3) Test set distance f_2: The procedure **select** calls f_2 to find a candidate test case that is farthest away from the set of test cases already selected. Here lies the problem of how to define "farthest away". We have formulated several strategies: Following Chen et al. [3], we choose a test case that has the largest minimum distance with the test cases in a given set, as shown by equation (1) in the definition of f_2. Following Ciupa et al. [8], we choose a test case that has the

largest average distance with the test case in a given set, as shown by equation (2) in the definition f_2. Another possible strategy is to choose a test case that has largest maximum distance with the test cases in the candidate set, as in equation (3). There can be other strategies to measure the similarity between two sets of test cases, which will be interesting to be studied further.

4) Level of coverage information: Our techniques use coverage information to compute the distance between a test case in the prioritized set and a test case in the candidate set. There are at least three levels of coverage information: statement, function, and branch.

TABLE I. PRIORITIZATION TECHNIQUES CONSIDERED IN THE PAPER.

Ref.	Name	Descriptions	
T1	random	Random prioritization	
T2	total-st	Total statement	
T3	total-fn	Total function	
T4	total-br	Total branch	
T5	addtl-st	Additional statement	
T6	addtl-fn	Additional function	
T7	addtl-br	Additional branch	
Ref.	ART	Level of Coverage Information	Test Set Distance (f_2)
T8	ART-st-maxmin	Statement	Equation (1)
T9	ART-st-maxavg	Statement	Equation (2)
T10	ART-st-maxmax	Statement	Equation (3)
T11	ART-fn-maxmin	Function	Equation (1)
T12	ART-fn-maxavg	Function	Equation (2)
T13	ART-fn-maxmax	Function	Equation (3)
T14	ART-br-maxmin	Branch	Equation (1)
T15	ART-br-maxavg	Branch	Equation (2)
T16	ART-br-maxmax	Branch	Equation (3)

C. ART Prioritization Techniques

We propose nine ART techniques. The algorithm in Figure 2 is used as the basis. Each technique uses a different level of coverage information and a different test set distance function f_2. Table I lists all the ART techniques ($T8$ to $T16$) as well other prioritization techniques considered in this paper ($T1$ to $T7$).

IV. EMPIRICAL STUDY

A. Research Questions

We study four research questions in the empirical study.

RQ1: Are ART-based techniques more effective than random ordering? The answer to this question will help us decide whether it would be helpful to use ART rather than random ordering for test case prioritization (or even for tie-breaking when pairing up with other techniques).

RQ2: Do different levels of coverage information have significant impact on ART techniques? The answer to this question will help us decide which level of coverage information to use for ART techniques. It would be best if the least costly level can be used to achieve the most effective result. However, if it is not the case, we would like to know the best option on hand.

RQ3: Does different definitions of test set distance have

significant impact on ART techniques? The answer to this question will help us decide which distance metric to use for ART techniques. Similarly to RQ2, it would be best if the least costly metric could be found to be the most effective. Is this the case?

RQ4: Can ART techniques be as effective as coverage-based techniques? The answer to this question will tell us whether ART can be a promising technique for test case prioritization in that it can perform as effectively as traditional best prioritization techniques. Is such a technique low cost as well?

TABLE II. SUBJECT PROGRAMS.

Subject	Faulty Version	LOC[1]	Test Pool Size
tcas	41	133–137	1608
schedule	9	291–294	2650
schedule2	10	261–263	2710
tot_info	23	272–274	1052
print_tokens	7	341–342	4130
print_tokens2	10	350–354	4115
replace	32	508–515	5542
flex	21	8571–10124	567
grep	17	8053–9089	809
gzip	55	4081–5159	217
sed	17	4756–9289	370

B. Peer Techniques for Comparison

In our empirical study, we compare the ART prioritization techniques with random ordering and six existing coverage-based prioritization techniques (from [12]).

The total statement (total-st) test case prioritization technique sorts test cases in descending order of the total number of statements covered by each test case. In case of a tie, it selects the involved test cases randomly. The total function (total-fn) and total branch (total-br) test case prioritization techniques are the same as total-st, except that it uses function and branch coverage information instead of statement coverage information [11].

The additional statement (addtl-st) prioritization technique selects, in turn, the next test case that covers the maximum number of statements 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 just selects one of them randomly. The additional function (addtl-fn) and additional branch (addtl-br) test case prioritization technique are the same as addtl-st, except that it uses function and branch coverage information instead of statement coverage information[11][12] [13].

C. Subject Programs and Test Suites

We use the Siemens suite of programs, downloaded from

<hr/>

[1] We use the tool "SLOCCount" (available at http://www.dwheeler.com/sloccount) to count the executable lines of code.

SIR [9] at http://sir.unl.edu, as subject programs. These programs were originally created to support research on data-flow and control-flow test adequacy criteria [15]. Since the Siemens programs are small, we also use four real-life UNIX utility programs with real and seeded faults (download from http://sir.unl.edu) in the experiment. Table II shows the descriptive statistics for all the subject programs.

We use a UNIX tool, gcov, to collect dynamic program coverage information for prioritization. Following [11], we exclude the faulty versions whose faults cannot be revealed by any test case as well as the faulty versions whose faults can be detected by more than 20% of the test cases. Besides, we also exclude those versions that are not supported by our experimental platform. All remaining faulty versions are used in the experiment.

D. Experimental Environment

We carry out the experiment on a Dell PowerEdge 1950 server serving a Solaris UNIX. The server is equipped with 2 Xeon 5355 (2.66Hz, 4 core) processors with 8GB physical memory.

E. Effectiveness Metrics

In this paper, we measure how quickly a test suite can detect faults. Following [13], we use APFD as the metrics for the rate of fault detection. *APFD* measures the weighted average of the percentage of faults detected over the life of the suite. Let *T* be a test suite containing *n* test cases and let *F* be a set of *m* faults revealed by *T*. Let TF_i be the first test case in the reordered test suite *T'* of *T* that reveals fault *i*. The APFD value for *T'* is given by the following equation from [13]:

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

F. Experiment and Discussions

The experiment involves seven small-sized Siemens programs and four real-life medium-sized UNIX utility programs. Following [20], we use the branch-coverage-adequate test suites provided by SIR to conduct the test case prioritization. There are about 1,000 small test suites and 1,000 large test suites. Since most of the small test suites can only kill a small portion (less than 20%) of all the faulty versions, it makes the comparisons among techniques much less significant. Hence, we adopt in our experiment the large test suites, which can kill about more than 55% of all faulty versions. For the UNIX programs, we generate 1000 test suites iteratively from the test pool: In each iteration, we randomly select a test case and add it to the suite as long as it can increase the branch coverage of the suite. The process stops when we have covered all the branches or when the new test case can no longer improve the branch coverage.

Since all the ART techniques are based on random selection, we repeat each of them 50 times to obtain averages that can portray typical performance. To reduce the huge computation cost in the experiment, we randomly select 20 suites from all the available 1000 test suites for each of the
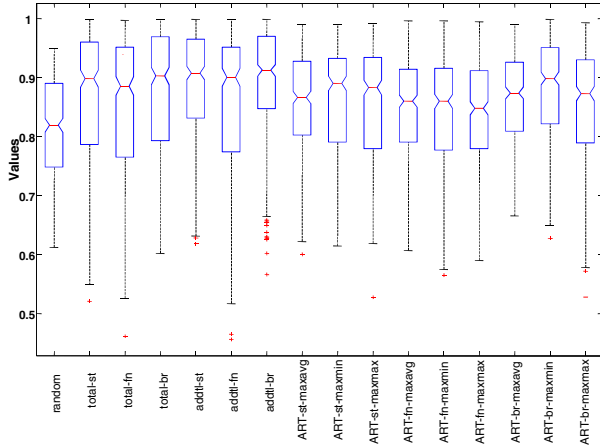
Figure 4. APFD Distributions for All Siemens Programs.

Siemens programs and UNIX programs. Thus, we conduct a total of 1000 prioritizations for each ART technique.

1) Is ART better than random? In this section, we analyze the data to answer the research question whether ART techniques are more effective than random ordering.

For each technique, we calculate the APFD results across all the faulty versions and draw box-and-whisker plots for the Siemens and UNIX programs, as shown in Figure 3 and Figure 4, respectively. For each box-whisker plot, the *x*-axis represents prioritization techniques and the *y*-axis represents their APFD values for all faulty versions. The lines in the boxes indicate the lower quartile, median, and upper quartile values. Boxes whose notches do not overlap mean that the medians of the two groups differ at the 5% significance level.

Figure 3 shows the box-whisker plots across all the Siemens programs. We observe that, in general, all the adaptive random prioritization techniques outperform the random ordering. Moreover, one can confidently conclude the medians of ART techniques differ from random ordering at the 5% significance level based on the non-overlapping of the notches.

Figure 4 shows the box-whisker plots for all the UNIX programs. We observe that ART prioritization performs better than both random ordering and the total techniques. The traditional total (statement, branch, function) prioritization techniques perform well on Siemens programs but not so on medium-size UNIX utility programs. (Similar results are also reported in [10][13].)

We also show the performance of individual Siemens and UNIX programs in Figure 5 and Figure 6, respectively. We observe similar results. For all the Siemens programs except schedule2, ART prioritization techniques perform better than random ordering significantly in terms of medians. For schedule2, we observe that three ART techniques (ART-st-maxavg, ART-br-maxavg, and ART-br-maxmin) perform better than random ordering while other ART techniques are comparable to random.

Having seen the box-whisker plots, we further conduct one-way analyses of variances (ANOVAs) to verify whether the means of the APFD distributions for different techniques

do not differ significantly. For all and each of the subject programs, the ANOVAs return a p-value much less than 0.01, which successfully rejects the null hypothesis at 1% significance level. For instance, the p-value for all Siemens programs is 0.00136. We will not discuss the p-values individually owing to space limit.

The four UNIX utility programs give similar results. From Figure 6, we find that the ART prioritization techniques in general perform better than random ordering. The p-values returned from ANOVAs also confirm that the means of various techniques differ significantly from random ordering.

We further conduct multiple comparisons to find those techniques whose means differ significantly from each other at the 5% significance level. The top line in Figure 7 represents the result of random ordering, which we use as a benchmark to compare with other techniques. The other solid lines represent the results of the techniques whose means differ significantly from random ordering, while the gray lines represents those of techniques comparable to random.

The results of the comparisons show that for all Siemens programs except schedule2, the means of APFD values for ART prioritization techniques are higher than that of random ordering. For the UNIX programs, when we compare ART techniques with random ordering, we find that all the techniques using "maxmin" (equation (1) for f_2) have higher mean APFD values than random ordering while other ART techniques are comparable to random.

Owing to space limitation, we will only show in Figure 7 the results of multiple comparisons across all Siemens and UNIX programs. Note here that the *x*-axis represents APFD values and the *y*-axis represents different prioritization techniques. We observe from Figure 7 that all ART prioritization techniques except ART-fn-maxmax have mean APFD values significantly higher than random ordering.

If we compare the means of all APFD values of individual ART techniques with that of random ordering, we find every ART technique can improve over random by 5 to 10%. At the same time, the best greedy coverage-based technique (namely, additional branch) can improve over random by up to 11%. Thus, the best ART technique (namely, ART-br-maxmin) and the best coverage-based technique can achieve almost the same improvements over random ordering.
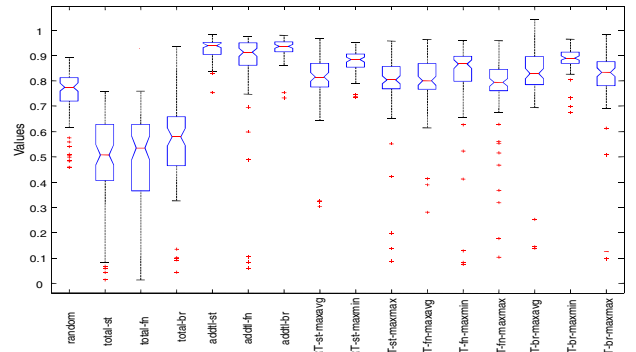

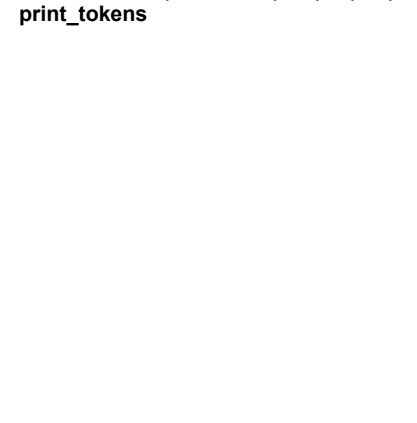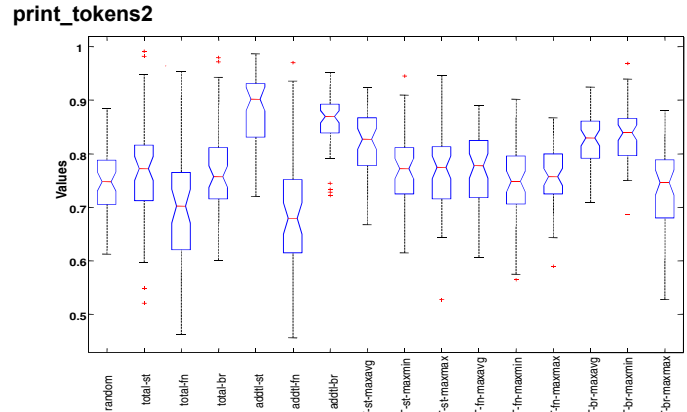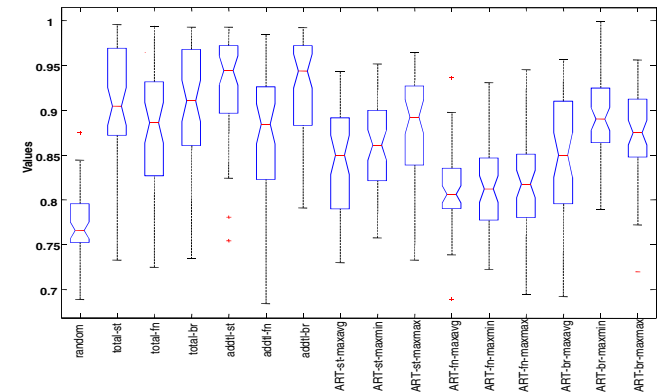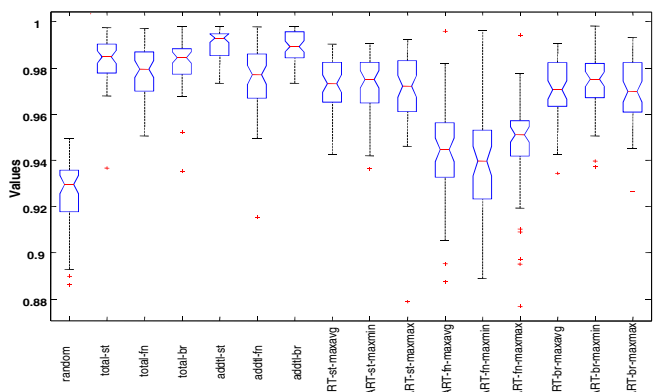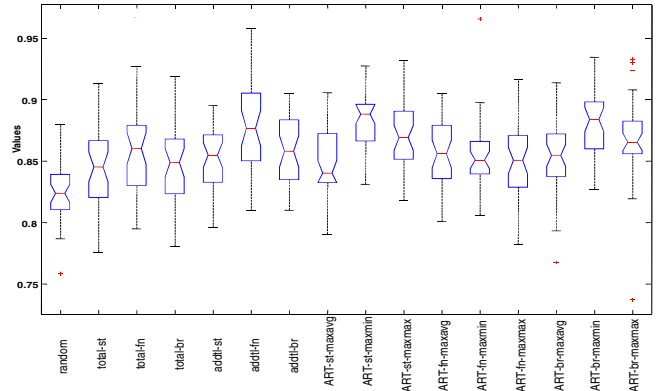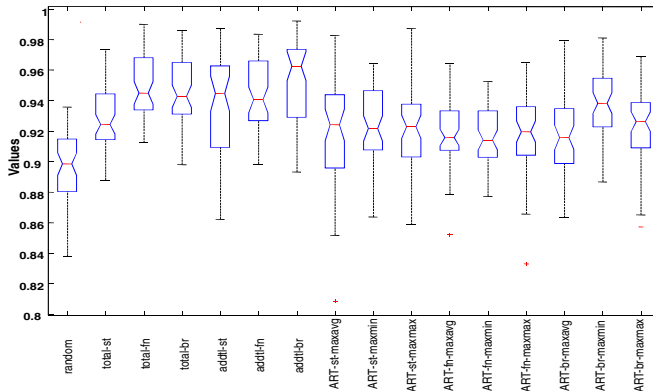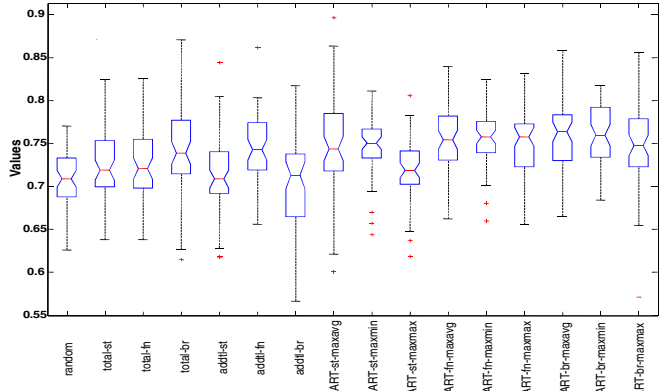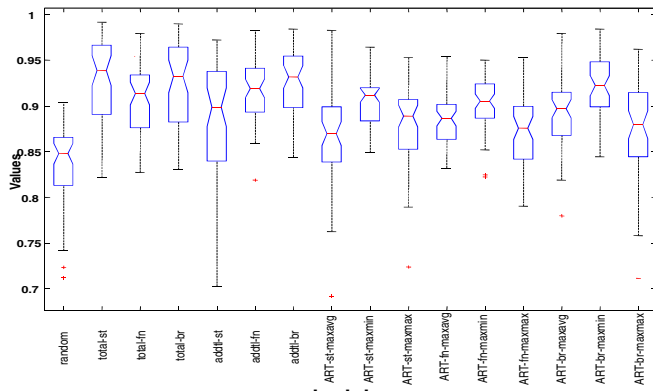Figure 3. APFD Distributions for All UNIX Programs.

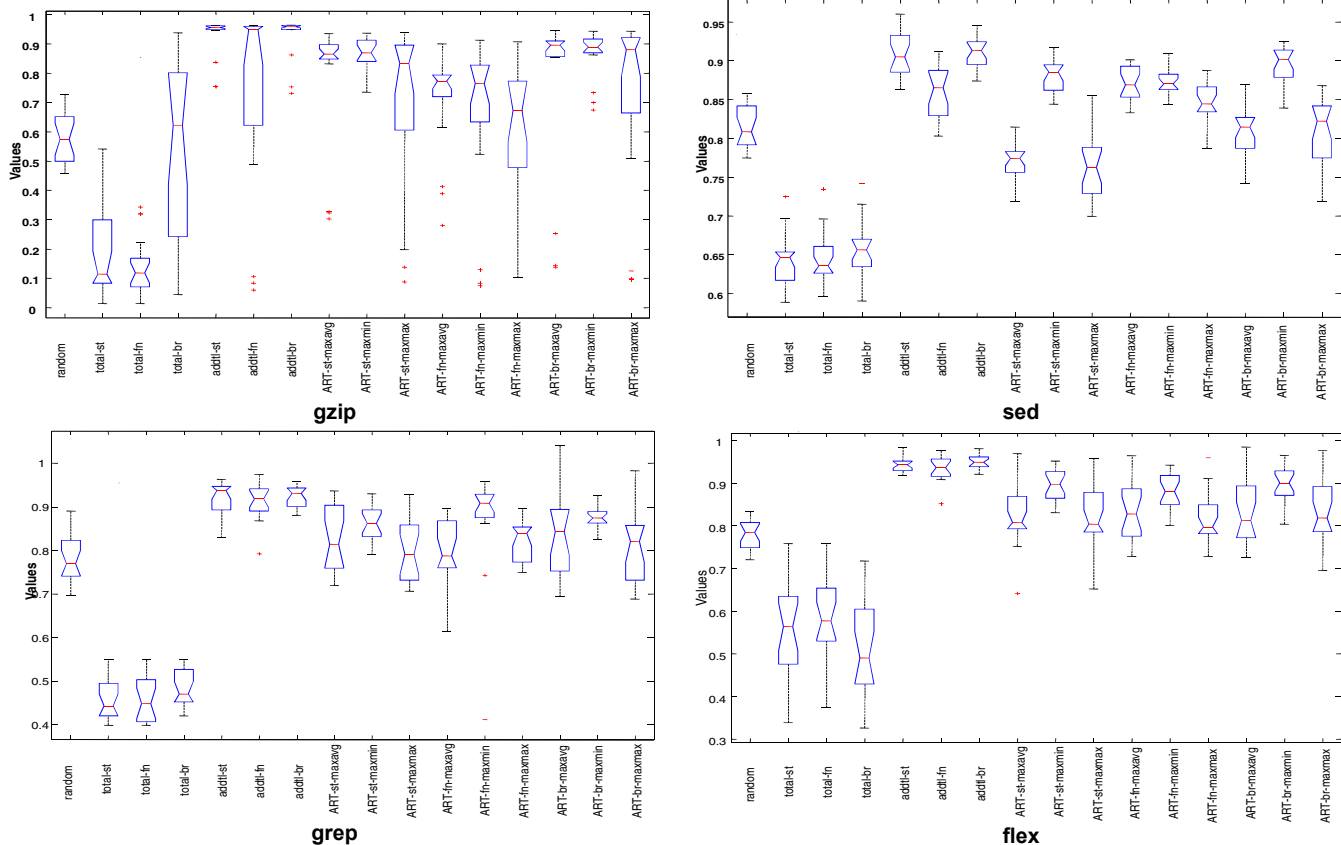Figure 5. APFD Distributions for Siemens Programs.

Figure 6. APFD Distributions for UNIX Programs.

To conclude, our experiment shows that, in general, the ART prioritization techniques do perform significantly better than random ordering across all subject programs.

2) Do different levels of coverage information have significant impact on ART techniques? In this section, we attempt to answer the research question whether different levels of coverage information have a significant impact on white-box ART techniques.

To compare the impacts of different levels of coverage information on APFD results of the ART prioritization techniques, we first categorize the techniques into three main groups according to their definitions of test set distances. The "maxmin", "maxavg" and "maxmax" groups, respectively, mean that we iteratively select a test case from the candidate set that maximizes its minimum, average, and maximum distances with the selected test set.

We then categorize each group into three subgroups according to the level of coverage information used, namely statement, function, and branch. For every subject program, we conduct multiple comparisons between each of the {statement:function, branch:function, branch:statement} pairs of subgroups. If the mean APFD value of first subgroup is higher (lower, respectively) than that of the second subgroup at 10% significance level, we put a ">" ("<") sign at the corresponding cell. In case there is no significant difference between their mean values, we put an "=" sign in the cell. The results are in Table III. For schedule2, for

instance, when the test set distance is "average", the ">" sign in the first data row indicates that the mean APFD values of statement-level ART techniques are statistically higher than those of function-level techniques.

From Table III, if we examine all the rows for "br:fn"[2], there are 12 ">" signs and 21 "=" signs, which means that branch-level techniques always perform better than or equal to function-level techniques. For the rows "br:st", we can see 29 "=" signs and 4 ">" signs, which means that branch- and statement-level techniques are generally comparable to each other. For "st:fn", there are 23 "=" signs, 9 ">" signs, and only one "<" sign, which means that statement-level techniques are general better than function-level techniques.

Furthermore, the impact of coverage information is more evident on small-sized programs than on medium-sized programs. The "maxmin" group is more sensitive to different levels of coverage information than the "maxavg" and "maxmax" groups.

In conclusion, different levels of coverage information do have impact on the ART prioritization techniques. In general, branch-level techniques are comparable to statement-level techniques and both of them are more effective than function-level techniques.

---

[2] That is, comparing the branch and function levels. Other labels in the same column can be interpreted similarly.
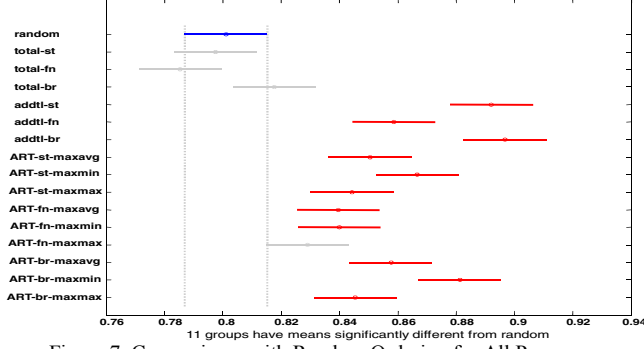
240

Figure 7. Comparisons with Random Ordering for All Programs.

TABLE III. COMPARING DIFFERENT LEVELS OF COVERAGE INFORMATION.

Program Size		Small							Medium-Sized			
Test Set Distance	Comparison	schedule	schedule2	tcas	tot-info	replace	print_tokens	print_tokens2	gzip	sed	flex	grep
maxmin	st:fn	=	=	=	=	>	>	>	=	>	=	=
	br:fn	=	>	=	>	>	>	>	>	>	=	=
	br:st	=	>	=	>	=	=	=	=	=	=	=
maxavg	st:fn	=	>	=	=	=	=	>	=	=	=	=
	br:fn	=	>	=	=	=	>	>	=	=	=	=
	br:st	>	=	=	=	=	=	=	=	=	=	=
maxmax	st:fn	=	=	<	=	>	>	>	=	=	=	=
	br:fn	=	=	=	=	=	>	>	=	=	=	=
	br:st	=	=	>	=	=	=	=	=	=	=	=

3) Do different definitions of test set distances have significant impacts on ART techniques? In this section, we would like to find out whether we should select a new test case from the candidate set to maximize the minimum, average, or maximum distance between the candidate test case and the set of test cases already selected.

Similarly to the last section, to compare the impacts of test set distance on APFD results of the ART prioritization techniques, we first categorize the techniques into three groups according to their levels of coverage information, namely statement, function, and branch. We then categorize each group into three subgroups according to their definitions of test set distance, namely "maxmin", "maxavg" and "maxmax". For every subject program, we conduct multiple comparisons between each of the {maxmin:maxavg, maxavg:maxmax, maxmin:maxmax} pairs of subgroups. Similarly to Table III, if the mean APFD value of first group is higher (lower, respectively) than that of the second group at 10% significance level, we put a ">" ("<") sign in the corresponding cell. In case there is no significant difference between their mean values, we put an "=" sign in the cell. The results are shown in Table IV.

If we examine all of the rows for "maxmin:maxavg" in Table IV, we find 11 ">" signs, 21 "=" signs and only one "<" sign, which means that "maxmin" test set distance is noticeably better than "maxavg". When we consider "maxmin:maxmax", we can find 11 ">" signs and 22 "="

signs, which implies that the mean APFD values for "maxmin" are always higher or equal to those for "maxmax". For "maxavg:maxmax", there are 27 "=" signs, 4 ">" signs, and 2 "<" signs, which implies that the mean APFD values of "maxavg" and "maxmax" are comparable. We also observe that the statement group and the branch group are more sensitive to changes in test set distances.

TABLE IV. COMPARING DIFFERENT DEFINITIONS OF TEST SET DISTANCE.

Program Size		Small							Medium-Sized			
Level of Coverage Info.	Comparison	schedule	schedule2	tcas	tot-info	replace	print_tokens	print_tokens2	gzip	sed	flex	grep
st	maxmin: maxavg	>	<	=	=	>	=	=	=	>	>	=
	maxavg: maxmax	>	>	>	=	<	<	=	=	=	=	=
	maxmin: maxmax	=	=	>	=	=	=	=	=	>	>	=
fn	maxmin: maxavg	=	=	=	=	=	=	=	=	=	=	>
	maxavg: maxmax	=	=	=	=	=	=	=	=	=	=	=
	maxmin: maxmax	>	=	=	=	=	=	=	=	>	>	=
br	maxmin: maxavg	>	=	=	>	>	>	=	=	>	>	=
	maxavg: maxmax	=	>	=	=	=	=	=	=	=	=	=
	maxmin: maxmax	>	>	=	>	=	=	=	=	>	>	>

In conclusion, for different test set distance definitions, the "maxmin" group has higher mean APFD values than either the "maxmax" or the "maxavg" group. The "maxmax" and the "maxavg" groups are comparable to each other. Ciupa et al. [8] use "maxavg" as the test set distance for applying ART to the test case generation in object-oriented software. Our findings above may indicate that the use of "maxmin" as test set distance may further improve the results. More experiments are required to confirm the conjecture.

Taking the discussions in last section and this section together, we find that the branch-level coverage information and "maxmin" test set distance definition are the best options for the respective features in ART test case prioritization techniques. Intuitively, therefore, the ART-br-maxmin should be the best ART prioritization technique.

A careful analysis of Figure 3 and Figure 4 will further confirm this conclusion. As shown in Figure 3, ART-br-maxmin is better than the entire function group of ART techniques and comparable to any other ART techniques. In Figure 4, ART-br-maxmin technique is comparable to the ART-st-maxmin and ART-fn-maxmin techniques, but better than any other ART techniques.
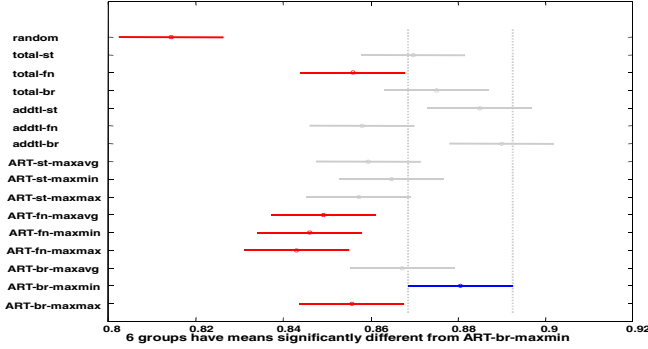
Figure 8. Multiple Comparisons for ART-br-maxmin on Siemens.

4) How does ART-br-maxmin compare with coverage-based techniques? From the discussions above, we know that ART-br-maxmin is the most effective technique in the proposed family of ART techniques. In this section, we are interested in how ART-br-maxmin performs when compared with the family of coverage-based prioritization techniques.
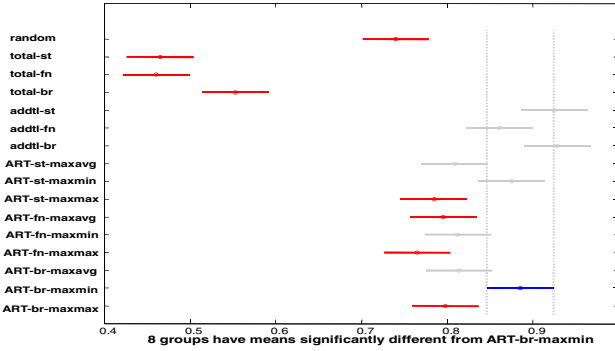

Figure 9. Multiple Comparisons for ART-br-maxmin on UNIX.

Basically, the ART prioritization techniques should only be compared with the total prioritization techniques rather than with the additional techniques. Current ART techniques have not incorporated the concept of "reset" into their algorithms, but simply select the farthest test case from the prioritized test set every time. It is possible that the prioritized test set has already achieved the maximal possible coverage for a given test suite, and all the remaining test cases have the same (maxmin, maxavg, or maxmax) distance from the prioritized test set. In such a scenario, it might be better to "reset" the prioritized test suite and continue to select test cases as if the prioritized test suite were empty. We plan to study the "reset" concept on top of the ART techniques and explore their effectiveness in future work. In this paper, we will continue to compare ART-br-maxmin with both the total and additional techniques.

We observe from Figure 5 and Figure 6 that ART-br-maxmin performs better than the total techniques on UNIX programs, and ART-br-maxmin is comparable to the total techniques for Siemens programs. We further use hypothesis testing to compare their means and confirm that ART-br-maxmin is comparable to the total statement and total branch techniques and better than the total function tech-

nique for Siemens programs, as shown in Figure 8, and that ART-br-maxmin performs significantly better than any total technique on UNIX programs, as shown in Figure 9.

One may further observer from Figure 5 and Figure 6 that ART-br-maxmin does not seem to perform as good as the additional techniques on both Siemens and UNIX programs. We argue that it is an illusion. When we use hypothesis testing to compare the means of ART-br-maxmin and the additional techniques, we can see the mean value of ART-br-maxmin has no significant difference from that of any additional technique, as shown in Figure 8 and Figure 9.

A detailed analysis of the results of multiple comparisons on each subject program also shows consistent findings: ART-br-maxmin is comparable to the best coverage-based prioritization technique in terms of APFD results.

TABLE V. TIME COMPARISONS OF DIFFERENT TECHNIQUES.

Program	schedule	schedule2	tcas	tot-info	replace	print_tokens	print_tokens2	gzip	sed	flex	grep
random	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
addtl-br	0.17	3.95	2.55	3.44	7.79	10.98	12.38	13.91	1.39	6.71	7.54
addtl-fn	0.28	4.70	3.94	4.85	8.91	20.52	17.61	19.79	1.78	6.49	6.97
addtl-st	0.45	7.04	5.27	8.87	15.61	22.44	25.97	43.28	2.79	22.87	21.72
total-br	0.01	0.08	0.05	0.05	0.12	0.24	0.24	0.71	0.12	0.48	0.69
total-fn	0.01	0.03	0.02	0.03	0.05	0.07	0.09	0.03	0.00	0.03	0.03
total-st	0.02	0.16	0.14	0.18	0.49	0.43	0.59	2.44	0.31	1.88	1.84
ART-br-maxmin	0.08	0.58	0.48	0.47	1.18	2.40	2.83	1.15	0.12	0.61	0.89
ART-br-maxavg	0.07	0.55	0.50	0.50	1.24	2.68	3.10	1.50	0.15	0.77	1.11
ART-br-maxmax	0.07	0.60	0.44	0.49	1.11	2.32	2.72	1.15	0.12	0.59	0.88
ART-fn-maxmin	0.03	0.27	0.19	0.25	0.65	0.75	1.07	0.48	0.01	0.12	0.21
ART-fn-maxavg	0.04	0.33	0.20	0.31	0.69	0.83	1.14	0.49	0.01	0.13	0.22
ART-fn-maxmax	0.04	0.31	0.19	0.31	0.69	0.83	1.08	0.48	0.01	0.13	0.21
ART-st-maxmin	0.11	1.25	0.87	1.30	2.61	4.26	5.77	2.78	0.31	1.88	2.02
ART-st-maxavg	0.10	1.33	1.01	1.35	2.94	4.32	7.30	3.94	0.36	2.46	2.61
ART-st-maxmin	0.11	1.23	0.80	1.24	2.59	3.86	5.44	2.84	0.31	1.86	2.04
Mean	0.10	1.40	1.04	1.48	2.92	4.81	5.46	5.94	0.49	2.94	3.06

5) Time-cost analysis: In this section, we further analyze the time cost of ART prioritization techniques and compare them with coverage-based techniques to help guide practical use. Table V presents the time cost (in seconds) for different prioritization techniques on different subject programs. We calculate the mean prioritization time across all techniques for every subject program and show each of them in the last row of the table. We observe that the additional techniques incur much more time cost than the mean prioritization time. The statement-level ART prioritization techniques have a time cost comparable with the mean. The branch- and function-level ART techniques, total techniques,

and random ordering always use much less time than the mean.

6) Conclusion: Based on the above discussions, we find that our family of techniques can be as effective and efficient as the coverage-based techniques. In particular, ART-br-maxmin can be comparable to the best coverage-based techniques (namely, the additional techniques) in terms of APFD results, and is much more efficient. Compared to the total technique, on average, ART-br-maxmin is more effective but slightly less efficient.

G. Threats to Validity

In this study, we use the Siemens suite and several UNIX utility programs as subject programs. All of them are either small or medium-sized programs. Further experiments on large programs may further strengthen the external validity of our findings. We choose only C programs owing to the limitation of our platform. Further investigations of other subject programs in different languages may help generalize our findings. For the Siemens suite, we use the branch-coverage-based test suites provided by SIR [9] to conduct our empirical study, but they may not represent all kinds of test cases in real-life practice.

Another threat to validity is the correctness of our tools. We use C++ to implement our tools for instrumentation, test suite prioritization, and results calculation. To minimize errors, we have carefully tested our tools to assure correctness.

Finally, we only explore white-box ART prioritization techniques in our controlled experimental study. A more complete study should further explore black-box ART test case prioritization techniques, that is, ART techniques anchored only in input information. Based on different definitions of input distances, we may come with another family of black-box ART test case prioritization techniques. We will leave the investigation of black-box test case prioritization techniques as future work.

V. RELATED WORK

In previous work, researchers have proposed many test case prioritization techniques. Do et al. [10] further investigate the impact of test suite granularity on the cost-benefits of various regression testing technologies. Wong et al. [28] proposed an approach for combining test suite minimization and prioritization to select cases based on the cost per additional coverage. Srivastava et al. [26] propose a binary matching system to compute the modified program in basic block level and prioritize test cases to maximally cover the amended program. Walcott et al. [27] propose a time-aware prioritization technique based on a genetic algorithm to re-order test cases under given time constraints. Li et al. [17] propose various search algorithms for test case prioritization. Their results show that genetic algorithms perform well, but greedy algorithms are surprisingly effective in increasing the code coverage rate. Both genetic algorithms and our ART techniques are effective in avoiding local maximums commonly faced by greedy algorithms. Since their goal for test case prioritization is to increase the code coverage rate while we aim at increasing the fault detection rate, our results are not directly comparable. We will perform further comparisons of ART and genetic techniques based on the same prioritization goals in future work.

Concolic testing [25] combines concrete and symbolic executions with a view to generate inputs to cover all feasible paths of a program effectively and efficiently. In concolic testing, random ordering only serves to generate initial inputs. Final useful test inputs are derived from the solutions of path constraints. In our ART techniques, however, prioritized test cases are selected from randomly generated candidate test sets. Hence, randomness takes a more active role in the ART techniques.

Researchers also study the problem of regression testing of service-oriented applications. Mei et al. [19] proposed a hierarchical of prioritization techniques for the regression testing of service-oriented business applications by modeling business process, XPath, and WSDL information. In [18], they also studied the problem of black-box test case prioritization of service-oriented applications based on the coverage information of WSDL tags.

Jiang et al. [16] study the problem of how well existing prioritization techniques support fault localization. They find random prioritization to be as effective as distribution-based prioritization techniques in supporting fault localization and can be a cost-effective option for practical use. In future work, we may explore whether adaptive random test case prioritization techniques can better support fault localization.

Adaptive random testing [2][3] improves the performance of random testing by adding guidelines to the test case generation process. Chen et al. [4] also propose the use of quasi-random sequences for testing, as they spread more evenly in a high dimensional space than random sequences. In restricted random testing [1], test cases are only allowed to be generated outside the exclusion zone so that they can be more evenly spread. Ciupa et al. investigated how to define the distance among objects for ART [6][7]. Their experimental results show that ART based on object distance can significantly increase the fault detection rate for object-oriented programs.

VI. CONCLUSION AND FUTURE WORK

Test case prioritization is a means to achieve target objectives in software testing by reordering the execution sequences of test suites. Many existing test case prioritization techniques use random selection to resolve tie cases. Paradoxically, the random approach for test case prioritization has a long tradition to be deemed as ineffective. Adaptive random testing (ART), which aims to spread test cases as evenly and early as possible over the input domain, has been proposed for test case generation. Empirical results have shown that ART can be 40 to 50% more effective than random testing in revealing the first failure of a program, which is close to the theoretical limit. In regression testing, however, further refinements of ART may be feasible in the presence of coverage information.

This paper proposes the first family of adaptive random test case prioritization techniques, and conducts an experiment to evaluate its performance. It explores the ART prioritization techniques with different test set distance definitions

at different code coverage levels rather than spreading test cases as evenly and early as possible over the input domain. The empirical results show that our techniques are significantly more effective than random ordering. Moreover, the ART-br-maxmin prioritization technique is a good candidate for practical use because it can be as efficient and statistically as effective as traditional coverage-based prioritization techniques in revealing failures.

In the future, we will investigate other test case measures and study beyond code coverage. Furthermore, we also want to extend our ART prioritization techniques to testing concurrent programs. Finally, as stated in the introduction, the use of random ordering to resolve tie-cases with existing greedy algorithms is deemed as ineffective. We would like to apply ART to resolve tie-cases in order to combine the merits of our techniques with other approaches.

REFERENCES

[1] K. P. Chan, T. Y. Chen, and D. P. Towey. Restricted random testing. In *Proceedings of the 7th European Conference on Software Quality* (*ECSQ 2002*), volume 2349 of Lecture Notes in Computer Science, pages 321–330. Springer, Berlin, Germany, 2002.

[2] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: the ART of test case diversity. *Journal of Systems and Software*, 2009. doi:10.1016/j.jss.2009.02. 022.

[3] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science: Proceedings of the 9th Asian Computing Science Conference* (*ASIAN 2004*), volume 3321 of M. J. Maher (*ed.*), Lecture Notes in Computer Science, pages 320–329. Springer, Berlin, Germany, 2004.

[4] T. Y. Chen and R. G. Merkel. Quasi-random testing. *IEEE Transactions on Reliability*, 56 (3): 562–568, 2007.

[5] T. Y. Chen and R. G. Merkel. An upper bound on software testing effectiveness. *ACM Transactions on Software Engineering and Methodology*, 17 (3): Article No. 16, 2008.

[6] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st International Workshop on Random Testing*), pages 55–63. ACM Press, New York, NY, 2006.

[7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the* 2007 *ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 84–94. ACM Press, New York, NY, 2007.

[8] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 71–80. ACM Press, New York, NY, 2008.

[9] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4): 405–435, 2005.

[10] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit test cases: an empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11: 33–70, 2006.

[11] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2000*), *ACM SIGSOFT Software Engineering Notes*, 25 (5): 102–112, 2000.

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

[13] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12 (3): 185–210, 2004.

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

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*), pages 191–200. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[16] B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *Proceedings of the 33rd Annual International Computer Software and Applications Conference* (*COMPSAC 2009*), volume 1, pages 99–106. IEEE Computer Society Press, Los Alamitos, CA, 2009.

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

[18] L. Mei, W. K. Chan, T. H. Tse, and R. G. Merkel. Tag-based techniques for black-box test case prioritization for service testing. In *Proceedings of the 9th International Conference on Quality Software* (*QSIC 2009*). IEEE Computer Society Press, Los Alamitos, CA, 2009.

[19] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th International Conference on World Wide Web* (*WWW 2009*), pages 901–910. ACM Press, New York, NY, 2009.

[20] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41 (5): 81–86, 1998.

[21] M. K. Ramanathan, M. Koyuturk, A. Grama, and S. Jagannathan. PHALANX: a graph-theoretic framework for test case prioritization. In *Proceedings of the* 2008 *ACM Symposium on Applied Computing* (*SAC 2005*), pages 667–673. ACM Press, New York, NY, 2008.

[22] G. Rothermel, S. G. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology*, 13 (3): 277–331, 2004.

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

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

[25] K. Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ ACM International Conference on Automated Software Engineering* (*ASE 2007*), pages 571–572. ACM Press, New York, NY, 2007.

[26] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the* 2002 *ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002*), ACM SIGSOFT Software Engineering Notes, 27 (4): 97–106, 2002.

[27] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. TimeAware test suite prioritization. In *Proceedings of the* 2006 *ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2006*), pages 1–12. ACM Press, New York, NY, 2006.

[28] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering* (*ISSRE 1997*), pages 264–274. IEEE Computer Society Press, Los Alamitos, CA, 1997.