Test Case Prioritization using Adaptive Random Sequence with Category-Partition-based Distance

Xiaofang Zhang
School of Computer Science and Technology, Soochow University
Suzhou, China
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
xfzhang@suda.edu.cn

Xiaoyuan Xie
State Key Laboratory of Software Engineering
Wuhan University
Wuhan, China
xxie@whu.edu.cn

Tsong Yueh Chen
Department of Computer Science and Software Engineering
Swinburne University of Technology
Hawthorn, Australia
tychen@swin.edu.au

***Abstract*—Test case prioritization schedules test cases in a certain order aiming to improve the effectiveness of regression testing. Random sequence is a basic and simple prioritization technique, while Adaptive Random Sequence (ARS) makes use of extra information to improve the diversity of random sequence. Some researchers have proposed prioritization techniques using ARS with white-box information, such as code coverage information; or with black-box information, such as string distances of the input data. In this paper, we propose new black-box test case prioritization techniques using ARS, and the diversity of test cases is assessed by category-partition-based distance. Our experimental studies show that these new techniques deliver higher fault-detection effectiveness than random prioritization, especially in the case of smaller ratio of failed test cases. In addition, in the comparison of different distance metrics, techniques with category-partition-based distance generally deliver better fault-detection effectiveness and efficiency; meanwhile in the comparison of different ordering algorithms, our ARS-based ordering algorithms usually have comparable fault-detection effectiveness but much lower computation overhead, and thus are much more cost-effective.**

Keywords—test case prioritization; adaptive random sequence; random sequence; catergory partition; string distance

I. INTRODUCTION

A main task in regression testing is the prioritization of test cases, which reorders the existing test cases to provide cost-effective testing. Various test case prioritization techniques have been developed based on different information. For example, history-based prioritization techniques use information from previous executions to determine test priorities; knowledge-based techniques use human knowledge to determine test priorities; and model-based techniques use a model of the system to determine test priorities.

Most of these techniques adopt white-box information, such as program source code coverage or fault detection history, to facilitate their prioritization operations. However, such information is obtained from historical versions, which may no longer be valid for the new version in regression testing. As a consequence, the prioritization result may be misleading for the new version.

Therefore, some researchers proposed prioritization techniques using black-box information of the new version. One intuitive idea is to have the top-ranked test cases as diverse across the input domain as possible. However, one difficulty of adopting such input domain information is about how to measure the diversity for non-numeric inputs. Ledru et al. proposed a prioritization technique, where the diversity of test cases is assessed using four classic string distance metrics[1]. This method requires to calculate the distance for each pair of test cases in the given test suite, which results in great computational overhead. Jiang and Chan proposed a family of input-based randomized local beam search techniques for test case prioritization, which adopts Adaptive Random Sequence (ARS) method to reduce the overhead in Ledru's method. To measure the diversity, they have also adopted edit distance in their experimental study[2].

Though the above studies proposed solutions to measure the diversity for non-numeric test cases, they both simply treat any non-numeric input as a string, and adopt the string distance to achieve this goal. However, this measurement may lead to various problems. Firstly, such a strategy may lead to the loss of some semantic information in the test cases. Without capturing the semantic information of test cases, the distances just depend on the text of the test cases may lead to unexpected conclusions. Secondly, this strategy may lead to the dependence of input structure information. Different kinds of string distance metrics have specific characteristics, and the choice of a proper distance metric should be according to the type of the input. Thus, not taking the input structure into account may result in an improper choice of distance metric. Intuitively speaking, in the context of prioritization, the distance metric is supposed to estimate the likelihood of two test cases having common failure behavior. In order to achieve this goal better, instead of simply measuring distances from the lexicographic information of the input, a more sophisticated metric is desired.

To address these problems, in this paper, we propose new prioritization techniques which adopt two new ARS-based ordering algorithms and a new distance metric. The two new ordering algorithms, namely offline and online ordering algorithms are based on ARS because the application of ARS can help reduce the high computation overhead of Ledru's

This work is partially supported by the National Natural Science Foundation of China (61103045, 61502329, 61502323, 61572375), and Collaborative Innovation Center of Novel Software Technology and Industrialization.

technique and preserve the test case diversity in the prioritized sequence. Furthermore, we adopt a new metric, namely category-partition-based distance, denoted as *CP* distance, which can keep the semantic and the structure information of a test case, and hence can better measure the distance between two test cases. CP distance has been proposed to measure the distance between two non-numeric inputs[3-4], using the concepts of categories and choices proposed by Ostrand and Balcer for the category-partition method[5]. It has been illustrated that CP distance can be used to deal with non-numerical inputs, and moreover it is a promising distance that taking the semantics of test cases into account[3-4].

In this paper, we are interested to investigate whether our ARS-based test case prioritization techniques with CP distance are cost-effective. In order to do so, we conducted experimental studies on 11 object programs. The experimental results have indicated that: (1) our ARS-based test case prioritization techniques with CP distance are significantly more effective than random ordering; (2) in the comparison of different distance metrics, techniques with CP distance generally deliver better fault-detection effectiveness and efficiency; (3) in the comparison of different ordering algorithms, our ARS-based ordering algorithms usually have comparable fault-detection effectiveness but much lower computation overhead, and thus are much more cost-effective.

The rest of the paper is organized as follows. Section II reviews some backgrounds. Section III introduces our new prioritization techniques, including the details of ARS-based ordering algorithms and CP distance metric. Section IV describes the experiment settings for the empirical study. Section V presents the experiment results to evaluate our new techniques. The related work is discussed in Section VI. Finally, it is the conclusion and future work.

II. Background

A. Test Case Prioritization Techniques

Test case prioritization is a process that reorders existing test cases so that those with the higher priority, according to some criteria, are executed earlier. Given a test suite, test case prioritization will find a permutation of the original test suite, aiming to maximize the objective function. Its formally definition is as follows[6]:

Given: T, a selected test suite; PT, the set of permutations of T; f, a function from PT to real numbers.

Problem: Find a $T' \in PT$ such that $(\forall T'')\ (T'' \in (PT \setminus \{T'\}))\ [f(T') \geq f(T'')]$.

To measure the performance of different prioritization techniques, *APFD* was proposed to measure the weighted average of the percentage of faults detected during the execution of the test suite. Let T be a test suite which contains n test cases, and let F be a set of m faults revealed by T. Let T' be the prioritized test sequence for T. Let TF_i be the sequence index of the first test case in T' which reveals fault i. The APFD for test suite T' could be given by the following equation[6]:

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

APFD value ranges from 0 to 1. An ordered test suite with higher APFD value has faster (better) fault detection rate than those with lower APFD values. In addition, an improved metric $APFD_C$ was proposed to measure fault detection rate that accounts for varying test case costs and fault severities[7].

Although there are various test case prioritization techniques designed according to different intuitions, random sequence is the most simple and basic strategy among them. It is simple in concept and is easy to apply even when source code, test history, specification or test input are unavailable or incomplete. Therefore, random sequence has been used as a benchmark for evaluating other prioritization techniques.

B. Adaptive Random Sequence

Adaptive Random Sequence (ARS) is basically a random sequence, which embeds the notion of diversity with the purpose of improving the performance of random sequence. ARS has been argued as a possible alternative to random sequence.

ARS is originated from the concept of Adaptive Random Testing (ART) [8]. ART is an enhancement of random testing through the concept of even spreading of test cases in the input domain. During testing, if a test case is found to be non-failure-causing, it is very likely that its neighbors will not reveal any failures. ART has been implemented using various notions of even spread[8-11], and it has been applied into many different types of programs[3-4,12-13]. There is a close relationship between ART and ARS. Firstly, in the context of test case generation, ART is a test case generator to select test cases from the pool of possible inputs. However, if ART exhausts the whole pool of possible inputs, the generated order can be treated as a prioritized order. It means that ART can be used as a test suite "prioritizor" to deliver a prioritized sequence. Technically speaking, the test sequence generated by ART is an adaptive random sequence (ARS), which embeds the concept of even spread across the specific input domain. Secondly, ART is based on random testing with the objective of revealing the failures as early as possible, which is consistent with the objective of prioritization. As a consequence, ARS can be applied in test case prioritization.

In summary, ART aims at using fewer test cases to detect the failures by spreading test cases as evenly as possible across the whole input domain, while ARS aims for the same objective by spreading test cases as evenly as possible across the input domain specified by the regression test suite.

III. Methodology

In test case prioritization, the technique should be able to differentiate test cases and assign higher priority to a test case which is most different compared to those already prioritized, because the diversity of test cases can help improve their fault detection ability[14-15]. As a sequence, our intuition of prioritization is trying to have the top-ordered test cases as diverse as possible.

In this section, we will provide our prioritization techniques based on two new ARS ordering algorithms and a new distance metric. Before presenting the details of our techniques, we give some definitions at first. Given a regression test suite $T = \{t_1, t_2, \ldots, t_n\}$ to be prioritized. Denote the subset of T which has *already been prioritized* as P; and the subset of T which has *not yet been prioritized* as NP. Obviously, $T=P \cup NP$. Let $PT = <p_1, p_2, \ldots, p_k>$ denote the ordered list of test cases in P. As the prioritization is processed, test case t is iteratively subtracted from NP and added into P. Correspondingly, the ordered list PT will be appended with the newly added test case t, that is $< p_1, p_2, \ldots, p_k, t>$, which is denoted as $PT^\wedge t$.

A. Ordering Algorithm

1) ARS-all

Generally speaking, after the prioritization is completed, the test case sequence is finalized, and then the regression testing is conducted according to the prioritized test cases until testing resources exhaust. Like majority of the existing test case prioritization techniques, our offline ordering algorithm ARS-all is performed before executing the test cases.

ARS-all aims at selecting a test case farthest away from all already prioritized test cases. Algorithm 1 shows the process of ARS-all[16]: Initially, we randomly select a test case from T (line 2 to 5). Then, we use FSCS-ART[8] to decide the next test case. The algorithm constructs the *candidate set CS* by randomly selecting k test cases from NP(line 9), and then a candidate from CS will be selected as the next test case if it has the longest distance to its nearest neighbor in P(lines 10 to 14). Here, the "distance" can be string distance, CP distance or any other distance to measure the difference between two test cases. The process is repeated until all the test cases are ordered in sequence. Since the size of NP will decrease while the sequence of test cases is being constructed, we use the minimum between k and $|NP|$ as the size of candidate set, denoted as min(k, $|NP|$)(line 8). Previous studies showed that k=10 is a reasonably fair setting to balance the testing effectiveness and the computation overhead of FSCS-ART[8].

2) ARS-pass

ARS-pass is an online ordering algorithm that requires online feedback information[16]. Technically speaking, the next prioritized test case depends on the previous execution results of the current version. Hence, the prioritization must be conducted interleaving with program execution. Suppose that previously prioritized and executed test cases have not revealed any failures. The next prioritized test case should be far away from them, aiming at revealing failures more quickly. Thus, the prioritized test case sequence is incrementally generated according to the results of executed test cases.

ARS-pass can be easily implemented using the concept of *forgetting*[17], which is also referred to as ART with selective memory [18]. Different from ARS-all where the set P is the already prioritized set, the set of P in ARS-pass will only memorize the non-failure-causing test cases using the online feedback information of program execution. By forgetting the failure-causing test cases and only focusing on the passed test cases, ARS-pass not only reduces the distance computation overhead, but also ensures that each new test case is far away from all already executed but non-failure-causing test cases.

```
Algorithm 1: ARS-all
Inputs: T: {t_1, t_2, …,} is a set of test cases
        k: the number of candidates
Output: PT: <p_1, p_2, …,> is a sequence of test cases
1.  Initialize: $PT \leftarrow \phi$, $P \leftarrow \phi$, $NP \leftarrow \phi$, $CS \leftarrow \phi$
2.  $t \leftarrow$ a test case randomly selected from $T$
3.  $PT \leftarrow PT$ ^ $t$
4.  $P \leftarrow P \cup \{t\}$
5.  $NP \leftarrow T - P$
6.  do
7.      Initialize: $CS \leftarrow \phi$
8.      $k \leftarrow \min(k, |NP|)$
9.      $CS \leftarrow$ randomly select $k$ test cases from $NP$
10.     for each candidate test case $t_j$, where $j$=1,2,…,$k$
11.         calculate its distance $d_j$ to its nearest neighbor in $P$
12.     end for
13.     find $t_b \in CS$ such that $\forall j$=1,2,…, $k$, $d_b \geq d_j$
14.     $t \leftarrow t_b$
15.     $PT \leftarrow PT$ ^ $t$
16.     $P \leftarrow P \cup \{t\}$
17.     $NP \leftarrow T - P$
18. while ( $NP \neq \phi$ )
19. return $PT$
```

B. Distance metric

Both ARS-all and ARS-pass are trying to help make top-ranked test cases as diverse as possible. As a reminder, the diversity can be measured in any distance metric. In this paper, we are interested in non-numeric inputs. As mentioned in Section I, previous studies used string distance metrics[1-2,16], including Hamming distance, Manhattan distance, Edit distance and Cartesian distance. All these distance metrics are based on lexicographic information and usually miss some semantic information of the input. For example, given a file name as the input, string distance metrics may treat it as regular string, without considering its semantic information. Therefore, in our prioritization techniques, instead of using traditional string distance metrics, we adopt the category-partition distance (CP distance) metric.

CP distance metric, originated from category partition method[5], has been proposed to measure the distance between two non-numeric inputs in ART[3-4]. Categories are defined as parameters and environment conditions determining the behavior of the software under test. Then, for each category, choices are defined as mutually exclusive sets of values which are expected to trigger similar computation and supposed to result in similar program behavior. Before computing CP distance, the categories and choices should be firstly identified for the program under test, then each test case will be rephrased with a list of choices and each choice corresponds to a particular category. Here, for a particular category, two test cases either have the same choice or have different choices. The number of categories that two test cases have different choices is used as the difference measure.

Obviously, the larger this number is, the less similarity these two test cases have and hence the more different program behaviors these two test cases will trigger.

For example, suppose an input x_i is associated with a list of categories $\mathbf{A}(x_i)$, then the corresponding choices for these categories form another list of $\mathbf{O}(x_i)$. The distance between two inputs x_1 and x_2 will be calculated as follows. First, we construct the set of distinct choices $\hat{\mathbf{O}}(x_1, x_2)$ for x_1 and x_2, that is $(\mathbf{O}(x_1)\cup\mathbf{O}(x_2)) \setminus (\mathbf{O}(x_1)\cap\mathbf{O}(x_2)))$. Next, we calculate the set of distinct categories $\hat{\mathbf{A}}(x_1, x_2)$ for x_1 and x_2. Finally, we have the distance between x_1 and x_2 as $|\hat{\mathbf{A}}(x_1, x_2)|$.

As an illustration, consider a simple object recognition system, which accepts three input parameters of color, shape and size. For each parameter, there could be various values, but the system can only distinguish an object color of red, yellow or blue, an object shape of sphere, cube or pyramid, as well as an object size of either large (i.e. size $>=1m^3$) or small (i.e. size$<1m^3$). For this program, we have all test cases associated with three categories, namely, $\mathbf{A}(x)$={Color, Shape, Size}; three choices for the Color category, namely, [red], [yellow] and [blue]; three choices for the Shape category, namely, [sphere], [cube] and [pyramid]; and two choices for the Size category, namely, [large] and [small]. Consider two program inputs x_1 and x_2, where x_1 is a light-red sphere of size 3.2 m^3, and x_2 is a blue sphere of size 2.7 m^3. Then, $\mathbf{O}(x_1)$={red, sphere, large} while $\mathbf{O}(x_2)$={blue, sphere, large}. In this case, $\hat{\mathbf{O}}(x_1, x_2)$={red, blue}, and $\hat{\mathbf{A}}(x_1, x_2)$={Color} because there is only one category--Color--in which x_1 and x_2 differ. So the distance between these two inputs is 1 using category-partition distance metric.

By incorporating CP distance metric into our proposed ARS-all and ARS-pass ordering algorithms, two new ARS-based prioritization techniques can be formed. These techniques can be used in different scenarios, and we are interested in whether these two techniques are cost-effective.

IV. Experiment Settings

A. *Investigated prioritization techniques*

As discussed above, the ordering algorithm and the distance metric are orthogonal to each other. Thus, in this study, we compare our methods (denoted as *T*2 that uses CP distance metric and ARS-all algorithm, and *T*3 that uses CP distance metric and ARS-pass algorithm) with other five commonly used methods, denoted as *T*1, *T*4, *T*5, *T*6 and *T*7 in Table I.

In Table I, *T*1 uses random sequence to do the prioritization, it is the benchmark technique; *T*4 uses CP distance metric and the algorithm proposed by Ledru (denoted as *Ledru*). Their algorithm firstly computes the distances for each pair of test cases to find the first test case with the maximum distance, and then it repeatedly chooses a test case which is most distant from the set of already ordered test cases, by searching the whole pool of all remaining test cases[1]; *T*5 uses Manhattan distance metric and ARS-all algorithm; *T*6 uses Manhattan distance metric and ARS-pass algorithm; and *T*7 uses Manhattan distance metric and *Ledru* algorithm.

As a reminder, we choose Manhattan distance metric (denoted as *Mht*) instead of investigating all the four distance metrics used by Ledru, because Manhattan distance metric was recommended as the best choice by Ledru's experiment results[1].

TABLE I. Investigated Prioritizaion techniques

Ref.	Name	Distance metric	Ordering algorithm
*T*1	Random	--	Random sequence
*T*2	CP-ARS-all	CP	ARS-all
*T*3	CP-ARS-pass	CP	ARS-pass
*T*4	CP-Ledru	CP	Ledru
*T*5	Mht-ARS-all	Manhattan	ARS-all
*T*6	Mht-ARS-pass	Manhattan	ARS-pass
*T*7	Mht-Ledru	Manhattan	Ledru

B. *Object programs*

To evaluate the performance of our prioritization techniques, we use two sets of object programs, totally 11 object programs. The information of these object programs are summarized in Table II.

TABLE II. Object programs

Name	Brief description	LOC	# mutants	# test cases	# categories	$R_f = \frac{\text{\# failed test cases}^*}{\text{\# test cases}}$ of *TS_C*
cal	Calendar display	163	11	162	3	0.28
comm	File comparator	144	27	754	8	0.25
look	File searcher	135	29	193	7	0.32
spline	Curve interpolation	289	16	700	10	0.43
uniq	File comparator	125	29	431	7	0.42
print_tokens	Lexical analyzer	483	7	4071	26	0.10
print_tokens2	Lexical analyzer	402	10	4071	26	0.26
replace	Search and replace tool	516	31(32)	5542	24	0.27
schedule	Scheduler	299	9	2650	34	0.23
schedule2	Scheduler	297	9(10)	2710	34	0.08
tot_info	Basic statistics	346	23	1052	19	0.52

*Note: A test case is failed if it can kill at least one mutant.

The first set of object programs used in this study is a set of five Unix utilities: *cal*, *comm*, *look*, *spline*, and *uniq*. These programs are part of object programs used for test suite reduction experiments by Wong et al.[19], and also part of object programs to analyze the code coverage achieved by ART[3]. For these Unix utilities, mutation faults have previously been generated by the automated C mutation tool, namely Proteum[20]. However, not all generated mutants are used, as some fail on every test case, whereas others are equivalent to the original program. Thus, we refine the initial set of mutants and choose some of them to use in this study. Firstly, we discard all mutants that are not killed by any test cases. We also discard all mutants that can be easily killed, that is, they can be killed by over 10% of all test cases. Finally, if there are some mutants that have exactly the same set of failure-causing inputs, we randomly select one of these mutants to use in the study[3]. After performing this refining process, we obtain the mutants and their corresponding test suites[3,19], denoted as *complete* test suite (*TS_C*).

The second set of object programs is selected from Siemens Suite from Software-artifact Infrastructure Repository (SIR). It is a widely adopted benchmark for various testing techniques. For these programs, we use the existing faulty versions present in the repository for comparison. Each program comes with a base version, several mutants and a test pool. For *replace* and *schedule2*, the given test suites cannot detect one of the mutants respectively, so we excluded these mutants from our experiments. In this study, we use the whole test pool as a test suite, also denoted as *complete* test suite (*TS_C*).

C. Ratio of failed test cases

The ratio of failed test cases, denoted as R_f, is defined as the ratio of the number of failed test cases to the total number of given test cases. Here, a test case is failed if it can kill at least one mutant. A large R_f means that a large portion of test cases are failed test cases and the mutants can be killed easily. However, in real-life testing, some faults can only be detected by a very small amount of test cases. Therefore, it is always expected that a good prioritization technique performs well for small-R_f scenarios. In this study, we are also interested in the performance of our method under different R_f values.

R_f of the *complete* test suites(*TS_C*) are summarized in the last column of Table II. Since R_f of *TS_C* is relatively large, apart from *TS_C*, we generate two additional test suites(*TS*_1 and *TS*_2) of smaller R_f, so that we can investigate the performance of prioritization techniques under different R_f values. For *TS*_1, R_f is 0.1 for Unix Suite and 0.02 for Siemens Suite. As for *TS*_2, R_f vary for each programs, the range of R_f is from 0.00067 for *printtokens2* to 0.05 for *cal*. Actually, *TS*_2 is the minimal set of test cases that can kill all of the mutants.

D. Measurement

In our study, we adopt APFD value to measure the effectiveness of a prioritization technique. Besides, we also investigate the time consuming that indicates the efficiency of a technique. We are interested in two following research questions.

RQ1: Are our ARS-based test case prioritization techniques with CP distance effective? In particular, what is the impact of R_f on the effectiveness?

RQ2: Are our ARS-based test case prioritization techniques with CP distance efficient?

The answers to these questions will help evaluate whether our prioritization techniques can be practical to be applied in the real-life testing.

V. Experiment Results

For each of the seven techniques in Table I, we prioritize three test suites, *TS*_C, *TS* _1 and *TS* _2 with all the 11 object programs in Table II. Since some of the prioritization techniques involve random selection, we repeat each of the above experiments for 100 times to obtain the average values.

A. Comparison of Effectiveness

To answer RQ1, we calculate APFD values for each object program on three test suites. Fig. 1(a) and Fig. 1(b) display the results in box-and-whisker plots for Unix utilities and Siemens Suite, respectively. In each box-and-whisker plot, there are three parts for *TS*_C, *TS*_1 and *TS*_2 test suites. For each test suite, we present the APFD distributions of seven prioritization techniques (*T*1 to *T*7).

To compare the impacts of different R_f on APFD values of each prioritization technique, we further draw scatter diagrams for each object program, as shown in Fig. 2. For each scatter diagram, the *x*-axis represents prioritization techniques and the *y*-axis represents their average APFD values of 100 trials. In each diagram, three graphical symbols correspond to *TS*_C, *TS*_1 and *TS*_2, respectively, and for each test suite, seven points correspond to seven prioritization techniques, respectively. Obviously, the higher point means the higher APFD value. While considering the distribution of three points for each technique, these three points cluster more closely indicates that the corresponding technique is more stable in different R_f values.

We conduct the comparisons of effectiveness in terms of APFD values from the following three aspects: (1) comparison with random ordering; (2) comparison between prioritization techniques using CP distance and Manhattan distance; and (3) comparison of ARS-all, ARS-pass and Ledru algorithms.

1) Comparison with random ordering

From Fig. 1(a)(b), we can observe that, all the other six prioritization techniques generally outperform random ordering in terms of median APFD values. Particularly, in this study, we focus on our new proposed CP-ARS-all and CP-ARS-pass. It is obvious that CP-ARS-all and CP-ARS-pass deliver higher APFD values than random ordering in most cases. Moreover, as shown in Fig. 2, with the smaller R_f, the difference between our proposed techniques and random ordering become much more significant. For example, for *spline*, in the case of *TS*_C, CP-ARS-all outperforms random ordering by 2.0% on average APFD value (0.9705 vs 0.9515), in the case of *TS*_1, CP-ARS-all outperforms random ordering by 19% on average APFD value (0.9067 vs 0.7323),

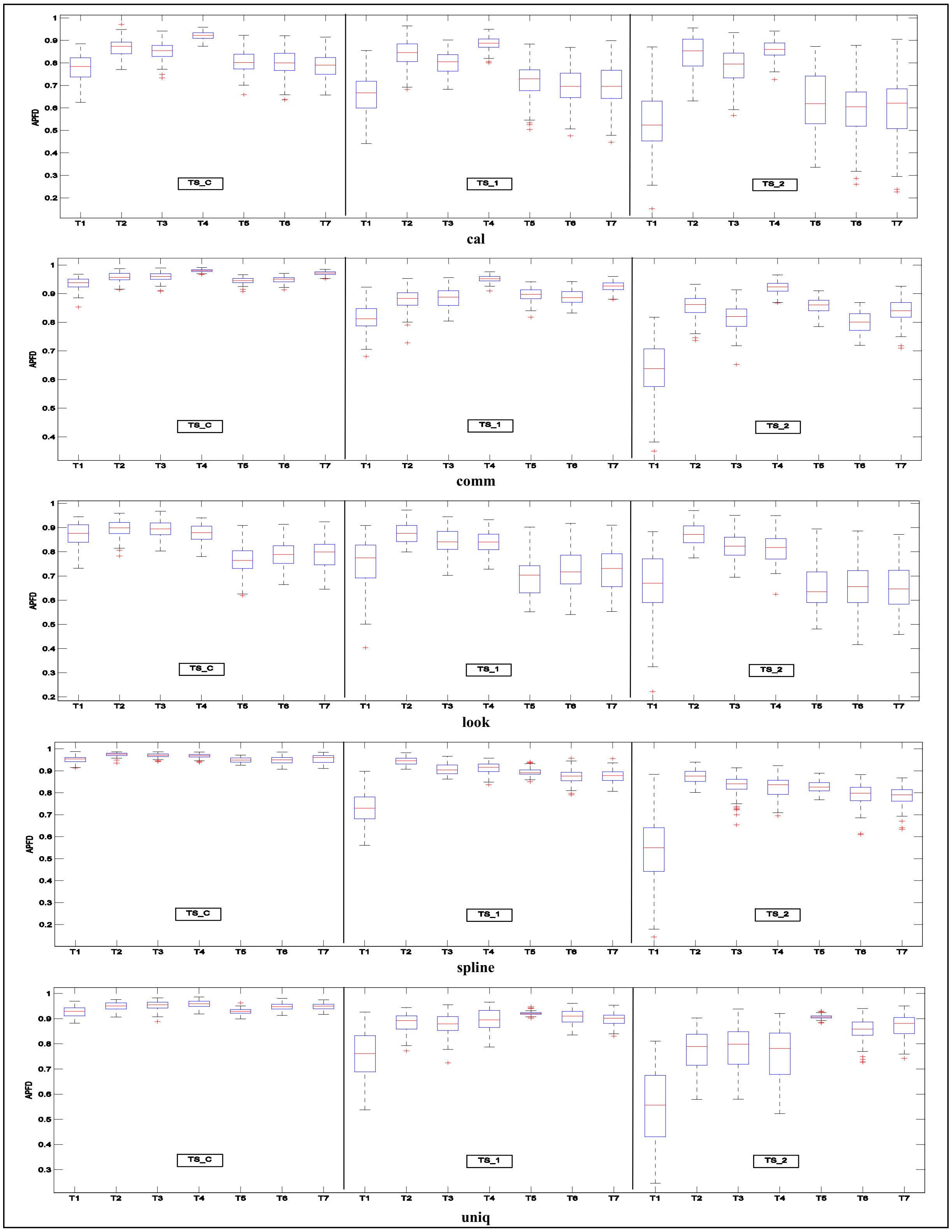

Figure 1(a). APFD distributions of three test suites for Unix utilities

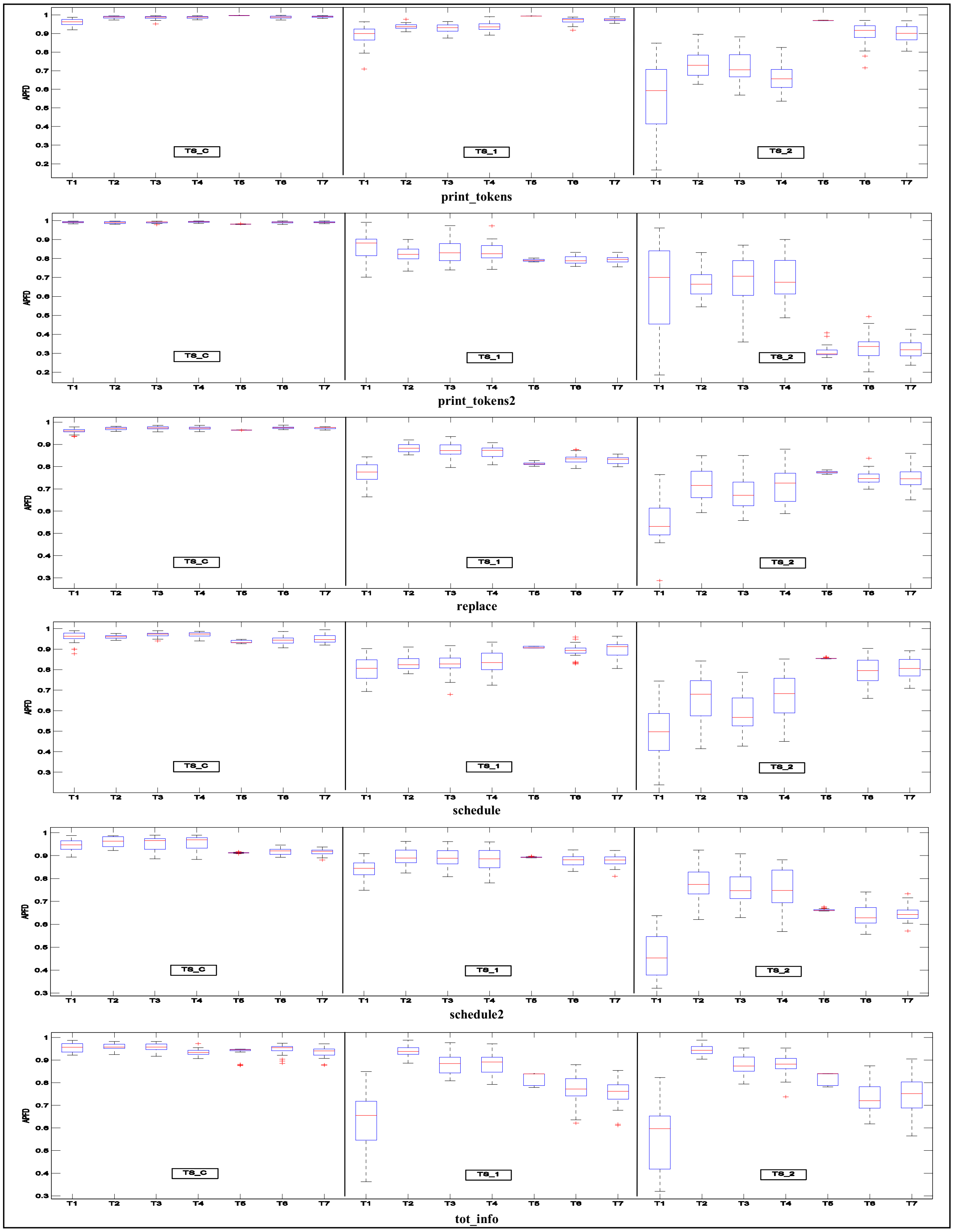

Figure 1(b). APFD distributions of three test suites for Siemens programs

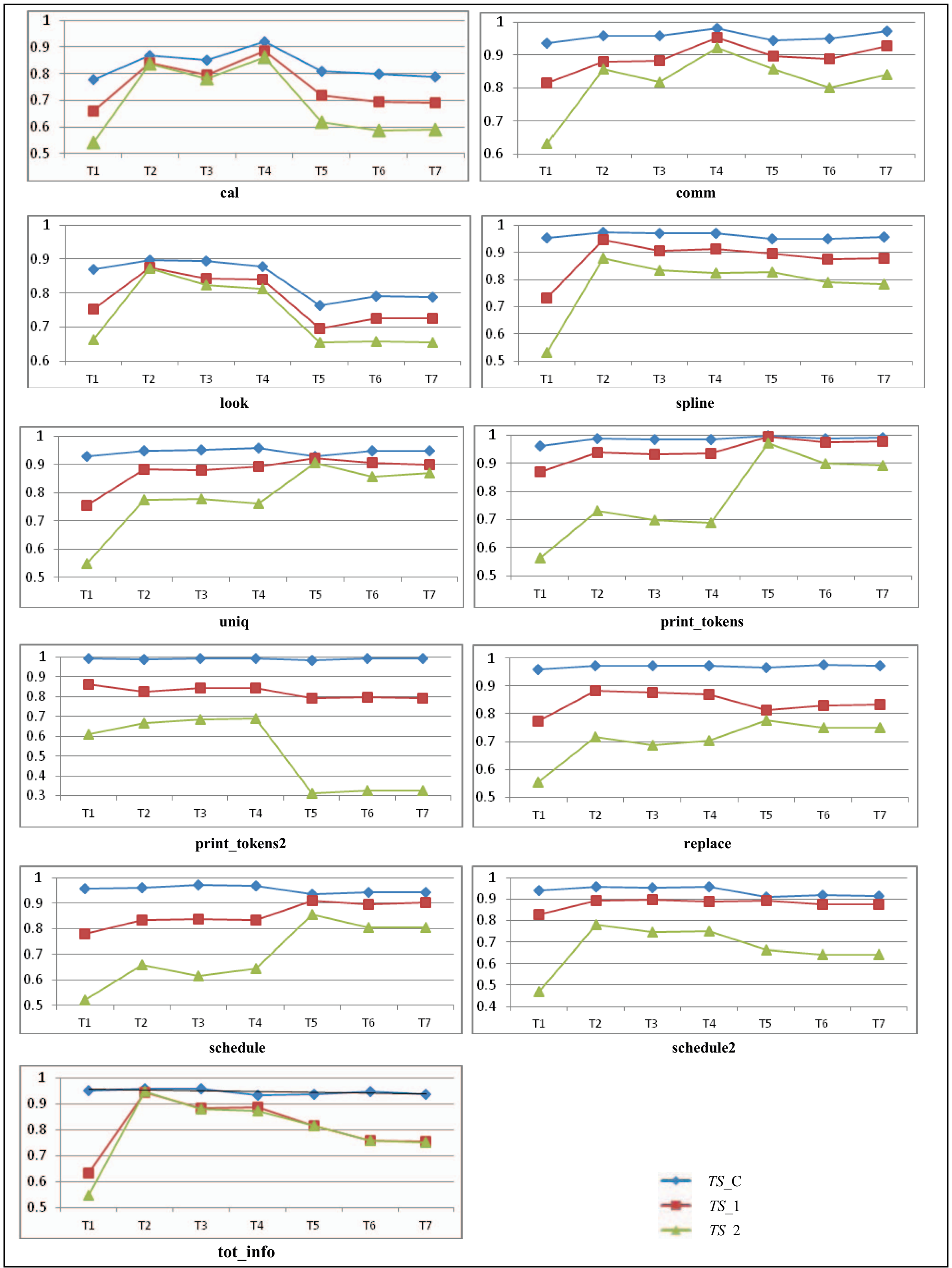

Figure 2. Average APFD values of three test suites for object programs

while in the case of *TS*_2, CP-ARS-all outperforms random ordering by 56% on average APFD value (0.8337 vs 0.5333). It is because that, in the case of smaller R_f, some faults are detected by a very small fraction of the test suite, and their probabilities to be detected early by a random ordering are decreased. Whereas, the ARS-based prioritization techniques with CP distance can still maintain the diversity of test cases to enhance the fault detection capability.

We further conduct multiple comparisons of CP-ARS-all, CP-ARS-pass and random ordering, as summarized in Table III. For each program, there are three test suites: {*TS*_C, *TS*_1 and *TS*_2}, two pairs of comparison subgroups: {CP-ARS-all vs. Random (*T*2 vs. *T*1), CP-ARS-pass vs. Random (*T*3 vs. *T*1)}. If the mean APFD value of the first subgroup is higher (lower, respectively) than that of the second subgroup at 5% 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. For *cal*, for instance, when the comparison subgroup is "CP-ARS-all vs. Random", and the test suite is "*TS*_C", the ">" sign indicates that the mean APFD value of CP-ARS-all is significantly larger than that of Random under this condition.

From Table III, we observe that, there are 56 ">" signs, 7 "=" signs and 3 "<" signs in a total of 66 cells, which means that CP-ARS-all and CP-ARS-pass techniques almost always perform better than or equal to random ordering. In addition, all of the three "<" signs appear in the case of *TS*_C and *TS*_1 while none of "<" sign appears in the case of *TS*_2. These observations indicate that the smaller R_f is, the bigger the difference is between our proposed techniques and random ordering.

In conclusion, our ARS-based test case prioritizations with CP distance generally outperform random ordering with respect to APFD values, especially in the case of smaller R_f.

2) *Comparison of distance metrics*

In this section, we conduct the comparison between prioritization techniques using CP distance and Manhattan distance. From Fig.1 and Fig.2, it is not very straightforward to get a conclusion; it seems that the results depend on the object program. Thus, we further conduct multiple comparisons on each object programs to verify whether APFD values for different distance metrics do differ significantly. For each program, there are three test suites: {*TS*_C, *TS*_1 and *TS*_2}, three ordering algorithms: {ARS-all, ARS-pass, Ledru}, two distance metrics: {CP, Mht} and totally three pairs of comparison subgroups: {CP-ARS-all vs. Mht-ARS-all (*T*2 vs. *T*5), CP-ARS-pass vs. Mht-ARS-pass (*T*3 vs. *T*6), CP-Ledru vs. Mht-Ledru (*T*4 vs. *T*7)}. The results of multiple comparisons are summarized in Table IV.

From Table IV, we can observe that: with regard to the comparison between prioritization techniques using CP distance and Manhattan distance, in a total of 99 cells, there are 67 ">" signs, 10 "=" signs and 22 "<" signs, indicating that prioritization techniques using CP distance metric achieve higher APFD values in most cases.

Furthermore, we can observe that, all of the 22 "<" signs appear in *uniq*, *print_tokens*, *replace* and *schedule*. It means that for these four programs, techniques with Manhattan distance have better APFD values than those with CP distance, especially in *TS*_2 with smaller R_f. We carefully examine the inputs of these four programs and find that, for each program, the test case with the longest string length is the failed test case. In other words, this kind of test case will easily be distinguished and given higher priority by prioritization with string distance. Manhattan distance metric in this study is so sensitive to the length of input string that the input with the longest or shortest length will be selected earlier. For example, for *print_tokens*, there are only 3 failed test cases remained in *TS*_2, and one of them is the test case that has the longest length among all of not-yet-prioritized test cases. Thus, this test case will be firstly prioritized to achieve the higher APFD value by the techniques with string distance. However, as for *print_tokens2*, which is another implementation of the same specification of *print_tokens*, test cases with longest or shortest length do not exist in the set of failed test cases that remained in *TS*_2. Therefore, in *print_tokens2*, the performances of techniques with string distance are worse than those of CP distance. These opposite results of *print_tokens* and *print_tokens2* indicate that the performances of prioritization techniques with string distance rely on the structure of inputs, usually ignore the semantics of inputs.

In conclusion, in our study, although sometimes Manhattan distance has its favorable scenarios to achieve higher APFD value, techniques with CP distance have better APFD values than those with Manhattan distance in most cases.

3) *Comparison of ordering algorithms*

In this section, we conduct the comparison of the following three ordering algorithms: ARS-all, ARS-pass and Ledru. These three prioritization algorithms can be applied using various distance metrics. Here, we compare them in the condition of CP distance and Manhattan distance. As shown in Fig.1 and Fig.2, that is *T*2 ~ *T*4 for CP distance subgroup, while *T*5 ~ *T*7 for Manhattan distance subgroup. Technically speaking, in each subgroup, Ledru should perform better than ARS-all and ARS-pass in terms of APFD values because Ledru algorithm searches the whole not-yet-prioritized test set, while ARS-all and ARS-pass just search the subset of not-yet-prioritized test set. Our experiment results are consistent with the theoretical analysis. In most cases, Ledru performs a little better; however, the differences between their APFD values are less than 5% according to Fig.1 and Fig.2.

We further conduct multiple comparisons of ARS-all, ARS-pass and Ledru on each object program to verify whether APFD values for different ordering algorithms do differ significantly. For each program, there are three test suites: {*TS*_C, *TS*_1 and *TS*_2}, two distance metrics: {CP, Mht} and three pairs of comparison subgroups: {Ledru vs. ARS-all, Ledru vs. ARS-pass, ARS-all vs. ARS-pass}.

For brevity, we just present the statistic results of multiple comparisons. We examine all the data for "Ledru vs. ARS-all", in a total of 66 cells, there are 31 ">" signs, 24 "=" signs and 11 "<" signs, which means that Ledru outperform ARS-all in 47% cases. As for "Ledru vs. ARS-pass", there are 28 ">" signs, 19 "=" signs and 19 "<" signs, which means that Ledru outperform ARS-pass only in 42% cases. When it turns to

TABLE III. COMPARISON WITH RANDOM ORDERING

Comparison	Test suite	cal	comm	look	spline	uniq	print_ tokens	print_ tokens2	replace	schedule	schedule2	tot_info
CP-ARS-all: Random	*TS*_C	>	>	>	>	>	>	=	>	>	=	=
	*TS*_1	>	>	>	>	>	>	<	>	>	>	>
	*TS*_2	>	>	>	>	>	>	=	>	>	>	>
CP-ARS-pass: Random	*TS*_C	>	>	=	>	>	>	>	>	>	=	<
	*TS*_1	>	>	>	>	>	>	<	>	>	>	>
	*TS*_2	>	>	>	>	>	>	=	>	>	>	>

TABLE IV. COMPARISON BETWEEN PRIORITIZATION TECHNIQUES USING CP DISTANCE AND MANHATTAN DISTANCE

Comparison	Ordering algorithm	Test suite	cal	comm	look	spline	uniq	print_ tokens	print_ tokens2	replace	schedule	schedule2	tot_info
CP vs. Mht	ARS-all	*TS*_C	>	>	>	>	>	=	=	=	>	>	>
		*TS*_1	>	=	>	>	<	<	>	>	<	>	>
		*TS*_2	>	>	>	>	<	<	>	<	<	>	>
	ARS-pass	*TS*_C	>	>	>	>	>	<	>	=	>	>	=
		*TS*_1	>	>	>	>	=	<	>	>	<	=	>
		*TS*_2	>	>	>	>	<	<	>	<	<	>	>
	Ledru	*TS*_C	>	>	>	>	>	<	>	>	>	>	>
		*TS*_1	>	>	>	>	<	<	>	>	<	=	>
		*TS*_2	>	=	>	>	<	<	>	<	<	>	>

"ARS-all vs. ARS-pass", there are 6 ">" signs, 44 "=" signs and 16 "<" signs, which means that ARS-all and ARS-pass have comparable APFD values in most cases. Regarding the impact of different sets of R_f values, it seems that there is no significant relationship between the performance of each ordering algorithm and R_f.

In conclusion, ARS-pass and ARS-all algorithms deliver comparable performance with Ledru algorithm in terms of APFD values with respect to two types of distance metrics.

Based on the above three aspects of effectiveness comparisons, we have addressed our first research question: our ARS-based prioritization techniques with CP distance, namely CP-ARS-all and CP-ARS-pass, deliver higher fault-detection effectiveness than random prioritization, especially in the case of smaller R_f. Compared with prioritization techniques using Manhattan distance, both of CP-ARS-all and CP-ARS-pass outperform in most cases. With respect to comparison of the ordering algorithms, ARS-pass and ARS-all obtain comparable performance with Ledru.

B. Comparison of Efficiency

As stated in the above, the efficiency is measured by the time consumed (in milliseconds) in the prioritization process. We have observed that the comparison of these techniques with *TS*_C, *TS*_1 and *TS*_2 are similar. Due to the page limit, we just present the data of *TS*_C as an illustration, which are summarized in Table V.

From Table V, it is very obvious that CP-ARS-all (*T*2) and CP-ARS-pass (*T*3) are efficient among the six techniques, in all 11 object programs. Firstly, with the same distance metric, ARS-all and ARS-pass have shown obvious advantage over Ledru by costing much less time, in all cases. For example, by using CP distance, in program *schedule*2, ARS-all and ARS-pass cost 1954.5ms and 1885.0ms, respectively; while Ledru costs 90394ms. Besides, ARS-pass consistently performs best among these three ordering algorithms. Secondly, with the same ordering algorithm, CP performs better than Mht in about 87%(29/33) cases. And in many cases, the advantage of CP is quite obvious. For example, with ARS-all, in program *tot-info*, it takes only 221.4ms by using CP distance; while it takes 6639.2ms by using Manhattan distance.

TABLE V. PRIORITIZATION TIME (IN MILLISECONDS)

Name	*T*2:CP-ARS-all	*T*3:CP-ARS-pass	*T*4:CP-Ledru	*T*5:Mht-ARS-all	*T*6:Mht-ARS-pass	*T*7:Mht-Ledru
cal	3.6	2.7	14.8	4.7	3.6	17.7
comm	111.3	81.9	1764.9	154.3	134.9	1678.7
look	6.1	4.6	26.1	11.4	10.1	31.2
spline	98.4	69.7	1249.2	116.5	94.1	1270.1
uniq	30.9	22.9	264.0	37.4	28.6	273.3
print_tokens	5650.8	5396.2	311891	6887.9	6567.7	302578
print_tokens2	4535.6	4145.1	311524	6943.8	6299.4	302364
replace	15805	14952	781098	27945	26809	771221
schedule	1867.5	1733.1	84414	6618.9	6371.0	85396
schedule2	1954.5	1885.0	90394	6919.9	6898.6	91452
tot_info	221.4	150.0	3764.1	6639.2	6588.2	10206

The reason of the first observation is that, our ARS-based test case ordering algorithms adopt the concept of FSCS-ART, which dramatically decreases the overhead by randomly selecting a subset of not-yet-prioritized set. Moreover, ARS-pass algorithm selects a subset of previously executed test set by using *forgetting* technique, thereby further reduces computation cost. As for the second observation, it is because that the overhead of CP distance calculation depends on the number of categories, which is usually less than the length of input string.

As a consequence, we have addressed our second research question and drawn the conclusion that our ARS-based test case prioritization techniques with CP distance are efficient, and CP-ARS-pass is the best choice in terms of computation time.

Based on the discussions on effectiveness and efficiency, the research questions are addressed. Our CP-ARS-all and CP-ARS-pass prioritization techniques are the cost-effective choices because they can save a lot of time cost but almost without effectiveness loss. Thus, our ARS-based prioritization techniques with CP distance are practical in the real life testing.

C. Threats to Validity

The major threat to internal validity is the correctness of our experiment environment. We use Java to implement the tools for test case replacement, failure matrix generation, distance calculation, prioritization algorithm, and results calculation and so on. Hence, we have carefully reviewed and tested the code to try to improve its quality.

There are several threats to external validity of this study. Firstly, we select five Unix utility programs and six Siemens programs as object programs in this paper. Although these programs have various characteristics, they are either small or medium-sized programs. Further experiments on other larger object programs may help generalize our findings. Secondly, in order to use CP distance, appropriate categories and choices need to be identified for the program under test, such an identification process is somewhat subjective. Finally, we choose three different sets of R_f values as the representatives of different scenarios in the real life testing, we should be careful when making a general statement.

The threat to construct validity involves the measurement. In this paper, we use the most commonly adopted effectiveness metric in prioritization, namely, APFD to measure the effectiveness of a prioritized test suite. However, other measures should probably be considered in further work.

Finally, since we focus on CP distance, we only compared with Ledru's ordering algorithm due to the time limit. We will leave the comparison with other black-box ARS-based test case prioritization techniques as future work.

VI. Related Work

In this section, we review the closely related work from the following three aspects: traditional test case prioritization, adaptive test case prioritization and CP distance.

Wong et al. firstly presented a test suite minimization and prioritization technique for regression testing[21]. Rothermel et al. used execution information acquired in previous test runs to define test case priority[6]. Li et al. proposed several non-greedy algorithms, and compared these non-greedy algorithms to Rothermel et al.'s greedy algorithms [22]. Jeffrey and Gupta presented an approach on the number of paths covered during test runs[23]. Saha et al. proposed a new approach that reduced regression test prioritization to a standard information retrieval problem[24]. Hao et al. presented a unified test case prioritization approach that encompasses both the total and additional strategies[25]. Evidently, all of these prioritizations require the test history information of the previous versions. In [26], Yoo and Harman performed a comprehensive survey on test case prioritization techniques. Recently, Henard et al. presented a comprehensive experimental comparison of several white-box and black-box test prioritization techniques. They found that diversity-based techniques perform best among the black-box approaches[27].

As for adaptive test case prioritization, Jiang et al. proposed the family of code coverage based adaptive random test case prioritization using Jaccard distance[28]. Similarly, Zhou et al. used Manhattan distance to measure the difference of code coverage[29], and he further applied ART to prioritize test cases based on execution frequency profiles using frequency Manhattan distance [30]. All of their empirical results showed that they are statistically superior to random sequence in detecting faults. Recently, Fang et al. proposed a similarity-based test case prioritization technique based on farthest-first ordered sequence, which assumed the availability of certain coverage information and execution frequency profiles[31].

Besides adaptive random test case prioritization based on white-box information, some researchers begin to focus on black-box information. Ledru's prioritization technique used string distances to measure the test case diversity, and hence solely depended on the input text information[1]. However, their algorithm needs to compute the distances for each pair of test cases, therefore their prioritized test sequence incurs expensive overhead. Jiang et al. proposed the input-based randomized local beam search techniques for test case prioritization[2]. Their techniques reduce the computation cost by controlling the number of branches in the exploration tree to be explored at any one round to a small number. By adopting the concept of candidate set and the technique of forgetting, we applied adaptive random sequence in test case prioritization to reduce the computational overhead of Ledru's technique and maintain the test case diversity as well. Our preliminary experimental studies showed that adaptive random test case prioritization techniques are cost-effective[16]. It is true that black-box information based adaptive random test case prioritization circumvent the difficulties in the use of white-box information, nevertheless, all the prioritization techniques mentioned above measure the diversity of test cases by using string distance of the text of the inputs.

In this paper, we use CP distance to measure the difference of test case and elaborate our test case prioritization techniques. The adoption of CP distance is not completely new. CP distance is originally from Category Partition Method (CPM) [5]. Traditionally, CPM is used to generate test cases. Recently, CPM has been adopted to facilitate the measurement of the distance between non-numeric inputs. A new metric, namely CP distance, has been proposed to expand the application of ART[3-4]. In [3], CP distance was also used to investigate whether ART can achieve higher code coverage than RT with the same number of test cases.

VII. Conclusion and Future Work

This paper reported the first attempt to use adaptive random sequence and CP distance in test case prioritization. Our prioritization techniques apply adaptive random sequence with the purpose of selecting the next test case farthest away from all already prioritized test cases, and then achieving even-spread of test cases across the input domain specified by the regression test suite. To measure the difference of test cases, we adopt CP distance metric. Different from string distance that only depends on the text of test case, CP distance captures significant semantic information to distinguish test case.

In this paper, we presented ARS-based prioritization techniques with CP distance, that is, CP-ARS-all and CP-ARS-pass. To evaluate the performance of our techniques, we carried out the experiments on two sets of object programs, and under three different R_f values. The results of experiments and statistical analyses provided an obvious conclusion that, CP-ARS-all and CP-ARS-pass outperform random sequence in terms of APFD values, which was used as a baseline. With regard to different ordering algorithms, CP-Ledru only has better APFD values than CP-ARS-all and CP-ARS-pass in some cases, whereas CP-ARS-all and CP-ARS-pass always use much less time than CP-Ledru. Furthermore, with regard to different distance metrics, all of the techniques with CP distance are more cost-effective than those with Manhattan distance. In a word, CP-ARS-all and CP-ARS-pass are recommended choices because of their cost-effectiveness.

In future, we will conduct more experiments on various object programs, and test suites with different types, sizes, languages, and other attributes. It would be interesting to investigate the favorable conditions of CP distance, and other types of distance metrics will be considered. Furthermore, there are various ART algorithms in the literature, different ART algorithms and other efficiency improvement techniques will be investigated. Finally, adaptive random sequence can be applied in many other different domains, it has been argued as a possible alternative to random sequence. The application of adaptive random sequence is a promising research topic.

REFERENCES

[1] Y. Ledru, A. Petrenko, S. Boroday and N. Mandran. “Prioritizing test cases with string distances,” *Automated Software Engineering*, 19(1): 65-95, 2012.

[2] B. Jiang and W.K. Chan. “Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization,” In *Proceedings of the 37th Annual International Computer Software and Applications Conference*, pp. 190-199, 2013.

[3] T. Y. Chen, F. C. Kuo, H. Liu, and W. E. Wong. “Code Coverage of Adaptive Random Testing,” *IEEE Transactions on Reliability*, 62(1): 226-237, 2013.

[4] A. C. Barus, T. Y. Chen, F. C. Kuo, H. Liu, R. Merkel, and G. Rothermel. “A Cost-Effective Random Testing Method for Programs with Non-Numeric Inputs,” *Accepted by IEEE Transactions on Computers*.

[5] T. J. Ostrand and M. J. Balcer. “The category-partition method forspecifying and generating functional tests,” *Communications of the ACM*, 31(6): 676-686, 1988.

[6] 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.

[7] S. Elbaum, A. G. Malishvesky and G. Rothermel. “Test case prioritization: A family of empirical studies,” *IEEE Transactions on Software Engineering*, 28(2):159-182, 2002.

[8] T. Y. Chen, F.-C. Kuo, R. Merkel, and T. H. Tse. “Adaptive random testing: The ART of test case diversity,” *Journal of Systems and Software*, 83(1): 60-66, 2010.

[9] K. Chan, T. Y. Chen, and D. Towey. “Restricted random testing: Adaptive random testing by exclusion,” *International Journal of Software Engineering and Knowledge Engineering*, 16(4): 553-584, 2006.

[10] T. Y. Chen, G. Eddy, R. Merkel, and P. Wong. “Adaptive random testing through dynamic partitioning,” In *Proceedings of the 4th International Conference on Quality Software*, pp. 79-86, 2004.

[11] J. Mayer. “Lattice-based adaptive random testing,” In *Proceedings of the 20th International Conference on Automated Software Engineering*, pp. 333-336, 2005.

[12] 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*, pp. 71-80, 2008.

[13] Y. Lin, X. Tang, Y. Chen, and J. Zhao. “A divergence-oriented approach to adaptive random testing of Java programs,” In *Proceedings of the 24th International Conference on Automated Software Engineering*, pp. 221-232, 2009.

[14] H. Hemmati, A. Arcuri and L. Briand. “Achieving scalable model-based testing through test case diversity,” *ACM Transaction Software Engineering Methodology*, 22(1), 1-42, 2013.

[15] T. Y. Chen, F. C. Kuo, D. Towey, Z. Zhou. “A revisit of three studies related to random testing,” *Science China Information Sciences*, 58 (5): 1-9, 2015.

[16] X. Zhang, T. Y. Chen and H. Liu. “An Application of Adaptive Random Sequence in Test Case Prioritization,” In *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering*, pp. 126-131 , 2014.

[17] K. Chan, T. Y. Chen, and D. Towey. “Forgetting test cases,” In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, pp. 485-492, 2006.

[18] H. Liu, F. Kuo, and T. Y. Chen. “Comparison of adaptive random testing and random testing under various testing and debugging scenarios,” *Software: Practice and Experience*, 42(8): 1055-1074, 2012.

[19] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. “Effect of test set minimization on fault detection effectiveness,” *Software: Practice and Experience*, 28(4): 347-369, 1998.

[20] M. E. Delamaro and J. C. Maldonado. “Proteum- a tool for the assessment of test adequacy for C programs,” In *Proceedings of the Conference on Performability in Computing Systems*, pp. 79-95, 1996.

[21] 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*, pp. 230-238, 1997.

[22] Z. Li, M. Harman, and R. Hierons. “Search Algorithms for Regression Test Case Prioritization,” *IEEE Transactions on Software Engineering*, 33(4): 225-237, 2007

[23] D. Jeffrey and N. Gupta. “Experiments with Test Case Prioritization Using Relevant Slices,” *Journal of Systems and Software,* 81(2): 196-221, 2008.

[24] R. K. Saha, L. Zhang, S. Khurshid, D. E. Perry. “An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes,” In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, pp. 268-279, 2015.

[25] D. Hao, L. Zhang, L. Zhang, G. Rothermel, H. Mei. “A Unified Test Case Prioritization Approach,” *ACM Transaction Software Engineering Methodology*, 24(2): 10:1-10:31, 2014

[26] S. Yoo and M. Harman. “Regression testing minimization, selection and prioritization: a survey,” *Software Testing Verification and Reliability*, 22(2): 67-120, 2012.

[27] C. Henard, M. Papadakis, M. Harman, Y. Jia, Y. L. Traon. “Comparing White-box and Black-box Test Prioritization,” In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*, pp. 523-534, 2016.

[28] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. “Adaptive random test case prioritization,” In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pp. 233-244, 2009.

[29] Z. Zhou. “Using coverage information to guide test case selection in adaptive random testing,” In *Proceedings of the 34th Annual International Computer Software and Applications Conference, 7th International Workshop on Software Cybernetics*, pp. 208-213, 2010.

[30] Z. Zhou, A. Sinaga, and W. Susilo. “On the Fault-Detection Capabilities of Adaptive Random Test Case Prioritization Case Studies with Large Test Suites,” In *Proceedings of the 45th Hawaii International Conference on System Sciences*, pp. 5584-5593, 2012.

[31] C. Fang, Z. Chen, K. Wu, and Z. Zhao. “Similarity-based test case prioritization using ordered sequences of program entities,” *Software Quality Journal*, 22(2): 335-361, 2014.