An Application of Adaptive Random Sequence in Test Case Prioritization

Xiaofang Zhang
School of Computer Science and Technology
Soochow University
Suzhou, China
xfzhang@suda.edu.cn

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

Huai Liu
Australia-India Research Centre for Automation Software Engineering
RMIT University
Melbourne, Australia
huai.liu@rmit.edu.au

Abstract—**Test case prioritization aims to schedule test cases in a certain order such that the effectiveness of regression testing can be improved. Prioritization using random sequence is a basic and simple technique, and normally acts as a benchmark to evaluate other prioritization techniques. 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 code coverage information that is normally related to the test execution history of previous versions. In this paper, we propose several ARS-based prioritization techniques using black-box information. The proposed techniques schedule test cases based on the string distances of the input data, without referring to the execution history. Our experimental studies show that these new techniques deliver higher fault-detection effectiveness than random prioritization. In addition, as compared with an existing black-box prioritization technique, the new techniques have similar fault-detection effectiveness but much lower computation overhead, and thus are more cost-effective.**

Keywords – test case prioritization; adaptive random sequence; random sequence; random testing; adaptive random testing

I. INTRODUCTION

Regression testing is used to ensure that changes to the program do not negatively impact its correctness. A main task in regression testing is the prioritization of test cases, which reorders the existing test cases to meet some performance goal, such as detecting the faults as early as possible. Previous studies on test case prioritization aim at providing earlier feedback to testers, and allow them to begin debugging earlier.

Various test case prioritization techniques have been developed, which are designed to achieve different objectives. Most of the test case prioritization techniques require white-box information or test history to facilitate their prioritization operations. Essentially, they use the information derived from the previous versions, such as program source code coverage or fault detection history. However, the white-box information and test history are not always available, and sometimes such a kind of information is incomplete or inaccurate, or even costly or difficult to obtain.

To address this problem, Ledru et al. [1] propose a prioritization technique based on string distances between test cases, which does not depend on the availability of white-box information or test history. Their technique is based on the concept of test case diversity in terms of four classic string distance metrics. However, their prioritization technique needs to calculate the distances for each pair of test cases in the given test suite. As a consequence, the computation overhead is considerably expensive.

Our study aims to reduce the high computation overhead while preserving the test case diversity in the prioritized sequence. We investigate a more cost-effective test case prioritization methodology using the black-box information of the program under test. Our prioritization techniques make use of the concept of adaptive random sequence to achieve diversity of test cases through the notion of evenly spreading across the input domain.

The rest of the paper is organized as follows. Section II introduces the background information of test case prioritization and adaptive random sequence. The previous work related to our study is discussed in Section III. Section IV gives the details of our new prioritization techniques. Section V reports our empirical study for evaluating the new techniques. The paper is finally summarized in Section VI.

II. BACKGROUND

A. Test Case Prioritization Strategies

Test case prioritization schedules test cases so that those with the higher priority, according to some criterion, are executed earlier in the regression testing process. Given a test suite, test case prioritization will find a permutation of the original test suite, aiming to maximize the objective function. There are various strategies based on different intuitions. 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.

To measure the performance of different prioritization strategies, APFD has been 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:

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

Among all prioritization strategies, random sequence is the most simple and basic strategy. It is simple in concept and is easy to apply even when source code, specification or test history is unavailable or incomplete. Therefore, random sequence has been used as a benchmark for evaluating other prioritization strategies.

B. Adaptive Random Sequence

Adaptive Random Sequence (ARS), originated from the concept of Adaptive Random Testing (ART) [2], is basically a random sequence embedding the notion of diversity. ARS has been argued as a possible alternative to random sequence.

ART is aimed to improve the fault-detection effectiveness of random testing through the concept of even spreading of test cases in the input domain [2]. It is motivated by the empirical observation that failure-causing inputs are frequently clustered into contiguous failure regions. In other words, if a test case is found to be non-failure-causing, it is very likely that its neighbors will not reveal any failures. Thus, preference should be given to select the input far away from the non-failure-causing inputs as the next test case. ART can be implemented using various notions of even spread, such as Fixed Size Candidate Set ART(FSCS-ART) [2], restricted random testing [3], ART by dynamic partitioning [4], lattice-based ART [5], and so on. In order to reduce the generation overhead for these algorithms, some general reduction techniques have been developed, such as clustering [6], mirroring [7], and forgetting [8]. Since its inception, ART has been applied into many different types of programs [6, 9].

There is a close relationship between ARS and ART. 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 input domain. Moreover, 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.

III. RALATED WORK

Test case prioritization schedules test cases with an intention to achieve some performance goal. Various test case prioritization techniques have been proposed using different intuitions.

Among these techniques, Rothermel et al. [10] emphasized using execution information acquired in previous test runs to define test case priority and they defined various techniques. Their techniques are shown to be effective at achieving higher values for APFD. Furthermore, Li et al. [11] proposed several non-greedy algorithms, including hill climbing algorithm and genetic algorithm. Evidently, all of these prioritizations require the test history information of the previous versions.

Similar to our investigation of test case prioritization by ARS, Jiang et al. [12] and Zhou et al. [13] proposed the family of adaptive random test case prioritization using code coverage. They used Jaccard distance and Manhattan distance respectively, to measure the difference of code coverage. The empirical results showed that they are statistically superior to the random sequence in detecting faults. Furthermore, Zhou et al. [14] applied ART to prioritize test cases based on execution frequency profiles using frequency Manhattan distance. Recently, Fang et al. [15] proposed a similarity-based test case prioritization technique based on farthest-first ordered sequence, which is similar to adaptive random sequence. However, these white-box methods assume the availability of certain coverage information or execution frequency profiles.

Ledru's prioritization technique used string distances to measure the test case diversity, and hence solely depended on the black-box information [1]. However, their algorithm 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. Therefore, their prioritized test sequence is deterministic but incurs expensive overhead.

IV. ARS-BASED TEST CASE PRIORITIZATION

In this section, we present our offline ARS-based test case prioritization algorithm, denoted as ARS-all, and online ARS-based test case prioritization algorithm, denoted as ARS-pass, respectively.

Before presenting our prioritization algorithms, we first give some definitions. Suppose $T = \{t_1, t_2, \ldots, t_n\}$ is a regression test suite with n test cases. A test sequence PT is an ordered list of test cases. If t is a test case, and $PT = <p_1, p_2, \ldots, p_k>$, we define $PT \hat{} t$ to be $< p_1, p_2, \ldots, p_k, t>$.

A. Offline Prioritization Algorithm: ARS-all

Majority of the existing test case prioritization techniques are applied offline. That is, 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.

Our offline prioritization algorithm ARS-all can be done in a batch-mode as other existing test case prioritization techniques. During the prioritization, two sets of test cases are maintained. One set is the *already prioritized set P*, the other set is the *not-yet-prioritized set NP*. Obviously, $T = NP \cup P$.

Our algorithm ARS-all aims at selecting a test case farthest away from all already prioritized test cases, which is summarized in Fig. 1: Initially, we randomly select a test case from T. Then, we use FSCS-ART algorithm to decide the next test case. The algorithm constructs the *candidate set CS* by randomly select k test cases from NP, 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. The process is repeated until all the test cases are ordered in sequence. In this paper, let k=10 according to the previous studies [2].

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

Figure 1. The prioritization algorithm: ARS-all

B. Online Prioritization Algorithm: ARS-pass

Different from most existing prioritization techniques that are applied in a batch mode, ARS-pass requires online feedback information. 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 generated according to the results of executed test cases.

Our prioritization algorithm ARS-pass can be easily implemented using the concept of forgetting, which is also referred to as ART with selective memory [16]. The subset of *P* 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. As a consequence, the probability of failure detection will be increased.

Note that we have proposed ARS-all and ARS-pass to reduce the prioritization overhead, and they can be applied in different execution modes. Particularly, ARS-pass will further reduce the computation overhead and improve the diversity of test cases by using the online feedback execution information of the program version under test.

V. Empirical Study

In this section, empirical evaluations of real-life program are presented to assess the performance of our algorithms.

A. Peer Techniques for Comparison

Besides random sequence, we compare our prioritization techniques with the technique proposed by Ledru et al. [1], denoted as *Ledru*.

In this study, two classic distance metrics, namely Manhattan and Hamming distances, are used to measure the diversity of test cases. Given two test cases represented by strings *A* and *B*, the Manhattan distance between them is calculated as $\sum_{i=1}^{n}|\alpha_i - \beta_i|$, where α_i and β_i denote the ASCII code for each character in *A* and *B*, respectively, and *n* is the longer length of *A* and *B*. When *A* and *B* have different lengths, the shorter one will be filled with NULL characters, whose ASCII code value is 0. For example, given *A* = "ac" and *B* = "abd", the Manhattan distance between *A* and *B* is |97–97| + |99–98| + |0–100| = 101. Manhattan distance was recommended as the best choice for the Ledru prioritization technique [1]. Hamming distance is the simplest distance metric. Given two test cases *A* and *B*, the Hamming distance between them is calculated as the number of characters different in these strings. When *A* and *B* have different lengths, the shorter string will be completed by a list of NULL characters in order to have the same length of the longer string. For the previous example *A* = "ac" and *B* = "abd", their Hamming distance is 2.

Combining different prioritization algorithms and distance metrics, we have seven prioritization strategies as follows, summarized in Table I.

TABLE I. PRIORITIZAION TECHNIQUES

Ref.	Name	Algorithm	Distance
*T*1	Random	Random	--
*T*2	Ledru-M	Ledru	Manhattan
*T*3	Ledru-H	Ledru	Hamming
*T*4	ARS-all-M	ARS-all	Manhattan
*T*5	ARS-all-H	ARS-all	Hamming
*T*6	ARS-pass-M	ARS-pass	Manhattan
*T*7	ARS-pass-H	ARS-pass	Hamming

B. Research Questions

We study the following research questions in the empirical study.

RQ1: Do ARS-based techniques perform better than prioritization using random sequence?

RQ2: Are ARS-based techniques more cost-effective than Ledru technique?

C. Subject Program and Test Suites

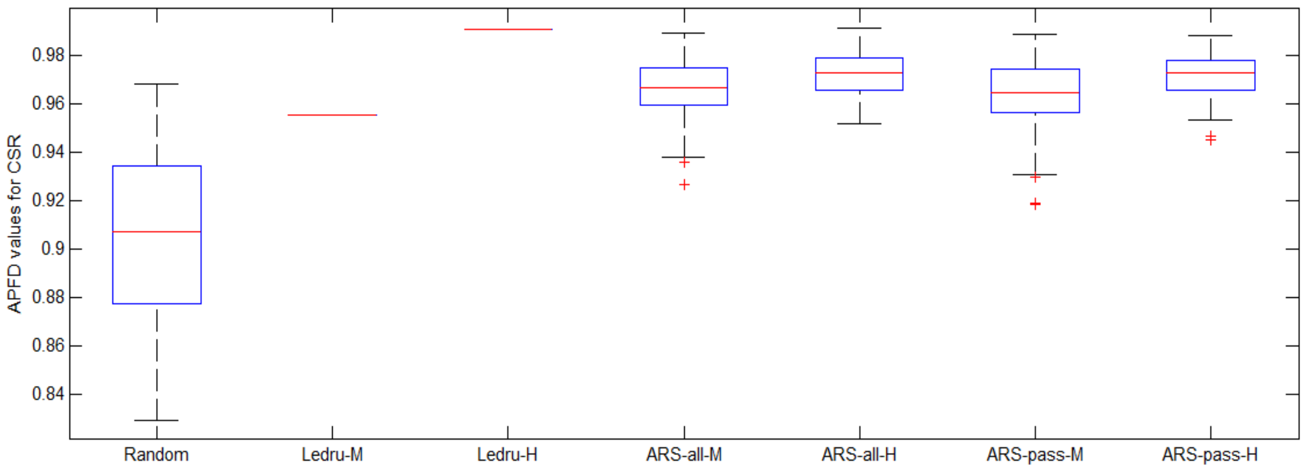

Figure 2. APFD values for test suite CSR

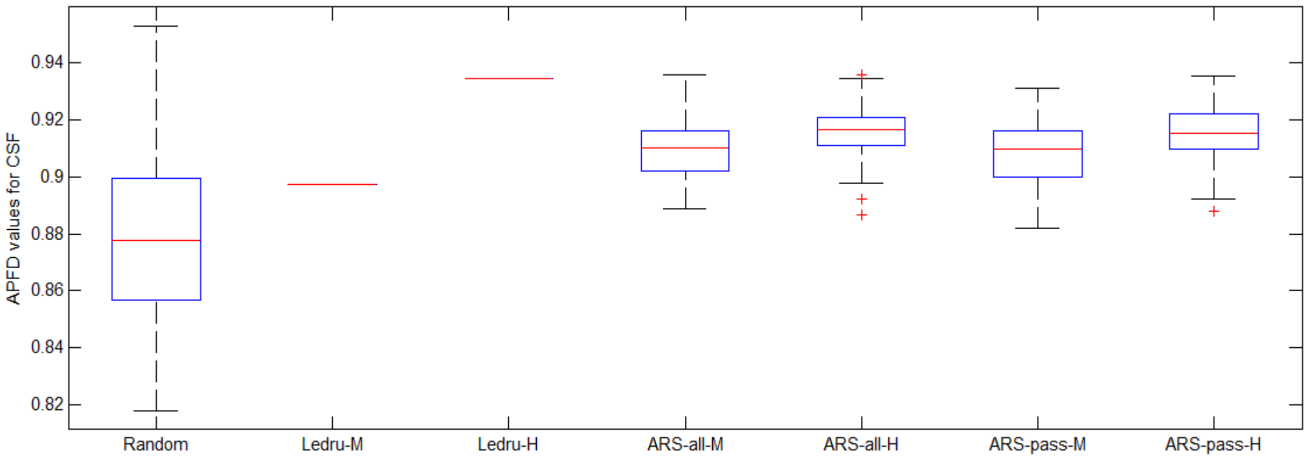

Figure 3. APFD values for test suite CSF

TABLE II. INFORMATION OF SUBJECT PROGRAM AND TEST SUITES

Application		*Crossword Sage*
Version		0.3.3
#Widgets		80
#Faults		14
#Test cases	**RT**	738
	FT	1278
#Detected faults	**RT**	12
	FT	14

Note: RT refers to Random Testing;
FT refers to Functional Testing.

We use a GUI application *Crossword Sage* as subject program. This subject program is free and open source in SourceForge. It presents unambiguous graphical user interfaces for testers to design the test cases. This application, written in Java, was widely used in some other studies [17-19]. Furthermore, this subject program contains real-life faults and exceptions.

Yang et al. [17] conducted an empirical study to compare random testing and functional testing in GUI testing. Thousands of random and functional test cases, in the form of Java scripts, had been created for this GUI application. With some necessary transformations, these existing test suites will be used for our prioritization. The basic information of our subject program and test suites is shown in Table II, and the details can be found in [17].

D. Experiment Design

For each prioritization strategy, we prioritize the existing two test suites, that is, *Crossword Sage Random* test suite and *Crossword Sage Functional* test suite, denoted as CSR and CSF, respectively. Thus, there are two sets of results for each prioritization strategy. The results include the prioritized sequence, the APFD value, the prioritization time cost, the detected faults report and so on. Since some of the prioritization strategies involve random selection, we repeat each of the prioritization strategies 100 times to obtain the averages.

E. Analysis of the Results

In this section, we analyze the experiment results to answer the research questions. As shown in Fig. 2 and Fig. 3, we calculate the APFD values for each strategy and draw box-and-whisker plots for the two test suites, respectively. For each box-and-whisker plot, the *x*-axis represents prioritization strategies and the *y*-axis represents their APFD values. The horizontal lines in the boxes indicate the lower quartile, median and upper quartile values. Furthermore, we present the average prioritization time cost (in milliseconds) of 100 trials for each strategy, summarized in Table III.

To address **RQ1**, we compare the APFD values between our ARS-based strategies and random sequence.

From Fig. 2 and Fig. 3, we observe that, for test suites CSR and CSF, all of our ARS-based strategies outperform the random sequence. We further conduct binomial t-tests for the APFD values for our ARS-based strategies and random sequence respectively. As shown in Table IV, all *p*-values are smaller than the significant level of 0.05. It means that all the null hypotheses (H_0: ARS-all-M/ ARS-all-H/ ARS-pass-M/ ARS-pass-H does not outperform Random) are rejected.

As a result, it is statistically significant that all of our ARS-based techniques perform better than random prioritization with respect to the APFD values.

TABLE III. INFORMATION OF PRIORITIZATION TIME (IN MILLISECONDS)

Technique	CSR	CSF
Random	0.20	0.34
Ledru-M	1632.67	10014.33
Ledru-H	1581.33	9480.00
ARS-all-M	56.90	194.51
ARS-all-H	58.76	194.67
ARS-pass-M	26.97	122.88
ARS-pass-H	29.08	120.14

To address **RQ2**, we compare the APFD values and prioritization time cost between ARS-based strategies and Ledru technique using these two test suites.

From Fig. 2, for test suite CSR, we observe that, with Manhattan distance, both ARS-all and ARS-pass significantly outperform Ledru. Whereas, for Hamming distance, Ledru-H has the best APFD value. Similar results also exist for CSF, as shown in Fig. 3. However, the differences between their APFD values are less than 5%. In other words, the performance of ARS-all, ARS-pass, and Ledru technique can be said to be comparable with respect to the APFD values.

For prioritization time cost (refer to Table III), it is obvious that all of ARS-based strategies use much less time than Ledru technique. In the best case, the computation time for ARS-based technique is about 1.23% of that for Ledru's technique (ARS-pass-M: 122.88 vs. Ledru-M: 10014.33 in CSF), while in the worst case the computation time for ARS-based technique is about 3.72% of that for Ledru's technique (ARS-pass-H: 58.76 vs. Ledru-H: 1581.33 in CSR). The reason for the dramatic cost reduction by our prioritization is that Ledru technique has to calculate all the distances of each pair of test cases before selecting next test case but not our ARS-based strategies.

Briefly speaking, given that our ARS-based techniques and Ledru technique have comparable APFD values, the former's lower overhead suggests that our ARS-based techniques are more cost-effective than Ledru technique.

TABLE IV. APFD COMPARISONS BETWEEN RANDOM SEQUENCE AND FOUR ARS-BASED TECHNIQUES

Algorithm(*x*)	Algorithm(*y*)	CSR		CSF	
		Mean Diff.(x-y)	***Sig.***	***Mean Diff.(x-y)***	***Sig.***
Random	ARS-all-M	-0.0611	0.0000	-0.0298	0.0000
Random	ARS-all-H	-0.0676	0.0000	-0.0367	0.0000
Random	ARS-pass-M	-0.0588	0.0000	-0.0285	0.0000
Random	ARS-pass-H	-0.0668	0.0000	-0.0356	0.0000

TABLE V. APFD COMPARISONS BETWEEN ARS-ALL AND ARS-PASS

Algorithm(x)	Algorithm(y)	CSR		CSF	
		Mean Diff.(x-y)	***Sig.***	***Mean Diff.(x-y)***	***Sig.***
ARS-all-M	ARS-pass-M	0.0016	0.2090	0.0013	0.3805
ARS-all-H	ARS-pass-H	0.0008	0.5531	0.0011	0.4136

We further compare the APFD values and prioritization time cost between ARS-all and ARS-pass. As shown in Fig. 2 and Fig. 3, ARS-all-M/H and ARS-pass-M/H have the similar APFD values. We conduct binomial t-tests of the APFD values for ARS-all and ARS-pass respectively, as shown in Table V. The online ARS-based algorithm, namely, ARS-pass, has comparable performance to the offline ARS-based algorithm, namely, ARS-all, in terms of the APFD values.

Next, we discuss their prioritization time cost. From Table III, we observe that, by making use of online feedback information of execution results and only focusing on the passed test cases, ARS-pass uses much less computation time. The computation time for ARS-pass is about 48% and 60% of that for ARS-all in CSR and CSF, respectively. Please note that, there are 302 passed test cases and 436 failed test cases in CSR while 749 passed test cases versus 529 failed test cases in CSF. In fact, the ratios of the time cost are consistent with the ratios of passed test cases in these two test suites.

In conclusion, ARS-pass has the comparable APFD values with ARS-all while using less computation time.

As explained in the above discussions, our adaptive random sequence outperforms random sequence and furthermore is cost-effective to apply in practice.

F. *Threats to Validity*

The major threat to internal validity is the potential faults in our tools. We use Java to implement the tools for GUI test case transformation, distance calculation, prioritization algorithm, and results calculation. Actually, we have carefully tested our tools on small examples to assure correctness.

Threats to external validity correspond to the subject programs and test suites. For the space limit, although we have carried out some other case studies, we just present the empirical study results of one GUI application, which is open source and widely used in several research works. Moreover, the used two test suites cannot sufficiently represent the real-life situations. Further experiments on other subject programs and test suites in different types and languages may help generalize our findings.

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

There is little threat to conclusion validity. A large number of experimental runs have been executed to get reliable average APFD values for the prioritization strategies involving (adaptive) random sequences. Statistical tests were conducted to validate the statistical significance of our experimental results.

VI. Conclusion and Future Work

This paper reported the first attempt to use adaptive random sequence and black-box information in test case prioritization. It is easy to implement, and it just requires the information about inputs. We carried out the experiments on an open source GUI application. The results of experiments and statistical analyses provided clear evidence that our ARS-based techniques are cost-effective in prioritizing test suites. Their performances of APFD are better than that of the random sequence, which was used as a baseline. Moreover, our techniques are much more cost-effective than Ledru technique. The ARS-pass algorithm achieves a good balance between APFD effectiveness and efficiency by using the online feedback information.

In future, we will conduct more experiments on various subject programs and test suites with different types, sizes, languages and other attributes. Furthermore, other efficiency improvement techniques for ART and other types of distance calculation will be considered. Finally, adaptive random sequence with different objectives to conduct prioritization is a promising research topic.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China (61103045), China Scholarship Council and Australian Research Council (DP 120104773). We would like to thank the authors of reference [17] to provide the test suites for our experiments.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] T. Y. Chen, F. Kuo, R. Merkel, and S. Ng. “Mirror adaptive random testing,” *Information and Software Technology*, 46(15): 1001–1010, 2004.

[8] 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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] C. Fang, Z. Chen, K. Wu, and Z. Zhao. “Similarity-based test case prioritization using ordered sequences of program entities,” accepted by *Software Quality Journal*, published online Nov. 2013. DOI: 10.1007/s11219-013-9224-0.

[16] 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.

[17] W. Yang, Z. Chen, Z. Gao, Y. Zou, and X. Xu. “GUI testing assisted by human knowledge: Random vs. functional,” *Journal of Systems and Software*, 89(3):76-86, 2014.

[18] A. Memon, M. Pollack, and M. Soffa. “Hierarchical GUI test case generation using automated planning,” *IEEE Transactions on Software Engineering*, 27 (2):144–155, 2001.

[19] A. Memon. “Automatically repairing event sequence-based GUI test suites for regression testing,” *ACM Transactions on Software Engineering and Methodology*, 18 (2), 4:1–4:36, 2008.