Test Case Prioritization for Compilers: A Text-Vector Based Approach

Junjie Chen[1,2], Yanwei Bai[1,2], Dan Hao[1,2*†], Yingfei Xiong[1,2†], Hongyu Zhang[3†], Lu Zhang[1,2], Bing Xie[1,2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[2]Institute of Software, EECS, Peking University, Beijing 100871, China
{chenjunjie,byw,haodan,xiongyf,zhanglucs,xiebing}@pku.edu.cn
[3]Microsoft Research, Beijing, 100080, China, honzhang@microsoft.com

***Abstract*—Test case prioritization aims to schedule the execution order of test cases so as to detect bugs as early as possible. For compiler testing, the demand for both effectiveness and efficiency imposes challenge to test case prioritization. In the literature, most existing approaches prioritize test cases by using some coverage information (e.g., statement coverage or branch coverage), which is collected with considerable extra effort. Although input-based test case prioritization relies only on test inputs, it can hardly be applied when test inputs are programs. In this paper we propose a novel text-vector based test case prioritization approach, which prioritizes test cases for C compilers without coverage information. Our approach first transforms each test case into a text-vector by extracting its tokens which reflect fault-relevant characteristics and then prioritizes test cases based on these text-vectors. In particular, in our approach we present three prioritization strategies: greedy strategy, adaptive random strategy, and search strategy. To investigate the efficiency and effectiveness of our approach, we conduct an experiment on two C compilers (i.e., GCC and LLVM), and find that our approach is much more efficient than the existing approaches and is effective in prioritizing test cases.**

I. Introduction

Software testing aims to guarantee software quality through the execution of test cases [1], [2], [3]. To improve the efficiency of software testing, test case prioritization [4] is proposed to schedule the execution order of test cases so that bugs can be detected as early as possible.

In the literature, there are many systematic and matured test case prioritization approaches [5], [6], [7], [8], [9], [10], [11], [12], [13]. However, most of these approaches [4], [14], [15], [5], [16] rely on some coverage information (e.g., statement coverage, branch coverage). Collecting such information consumes much extra effort. For example, Mei et al. [17] reported that the execution time of a program that records the coverage information (through instrumentation) is much larger than the execution time without instrumentation. Furthermore, in some practical testing scenarios, collecting such coverage information is infeasible. For example, when the software is tested for the first time, it is impossible to collect its testing information without executing its test cases. That is, these coverage-based approaches cannot be applied when coverage information is hard to collect.

*Corresponding author.
†Sorted in the alphabet order of the last names.

To prioritize test cases without coverage information, input-based prioritization [18], [19] is proposed by utilizing test input alone. In particular, two existing input-based prioritization approaches utilize either linguistic data (i.e., identifier names, comments and string literals) of test scripts or test-input text. However, when test inputs are programs (e.g., the test inputs of C compilers are C programs), our preliminary study shows that the time spent by these input-based approaches on prioritization is large[1]. That is, these existing input-based approaches can hardly be applied when test inputs are programs due to their efficiency problem.

To address the preceding issues, in this paper, we focus on prioritizing test cases for C compilers without coverage information. We choose C compilers due to the following reasons. First, the reliability of C compilers is very important because many safety-critical software are written in C. Second, C compiler testing is a typical and practical testing scenario for test case prioritization without coverage information, because collecting such information for C compilers is costly. Third, as C compiler testing consumes much time, it is more beneficial to execute fault-revealing test cases earlier. In the literature of C compiler testing, Yang et al. [20] spent three years on detecting 325 C compiler bugs, and Le et al. [21] spent eleven months on detecting 147 C compiler bugs. That is, test case prioritization is useful and important for C compiler testing.

In particular, we propose a novel text-vector based approach to test case prioritization, which transforms each test input into a text-vector representing its fault-relevant characteristics. In particular, our approach regards test inputs as text, and extracts their tokens reflecting fault-relevant characteristics so as to transform test inputs into a set of vectors. Then, after normalizing the values of elements in the set of vectors, our approach gives three prioritization strategies (i.e., greedy strategy, adaptive random strategy and search strategy) to prioritize test cases based on the set of vectors. Moreover,

[1]We applied the input-based approach proposed by Jiang and Chan [19] to a test suite and found that its time spent on prioritization is even far larger than the time spent on test suite execution. More details are referred to Section III-F1. Furthermore, the input-based approach proposed by Thomas et al. [18] achieves close efficiency to the basic string-based approach [18], which is slower than the approach proposed by Jiang and Chan [19]. Therefore, due to their high time cost for prioritization, neither of these approaches may be applied to software systems whose test inputs are programs (e.g., compilers).

to improve the prioritization efficiency, we apply principal component analysis (PCA) to optimize our approach with the adaptive random and search strategies. Although our approach is proposed to aid C compiler testing, it is not specific to C compiler testing and can be extended to other testing scenarios. More discussion on such extensions are referred to Section IV-B.

To investigate the efficiency and effectiveness of our approach, we apply our approach to two C compilers (i.e., GCC and LLVM), which cover all C compilers used in the literature of C compiler testing [20], [21], [22]. The experimental results show that our approach is much more efficient than the existing approaches and is effective in prioritizing test cases for C compiler testing. Furthermore, we also explore the impact of various factors (i.e., the PCA processing and distance formulae) on the effectiveness and efficiency of our approach. The results show that the PCA processing improves the efficiency of our approach by slight loss on effectiveness. Our approach tends to be stably effective and efficient no matter which distance formula is used.

To sum up, this paper has the following major contributions.

- We identify a practical and important test case prioritization scenario—the testing scenario of C compilers.
- We propose a novel text-vector based test case prioritization approach for C compilers.
- We conduct an experimental study that confirms the efficiency and effectiveness of our approach.

The remaining of this paper is organized as follows: Section II presents the details of our approach. Section III presents our experimental study and analyzes our experimental results. Section IV discusses the implicit precondition of test case prioritization and the extension of our approach. Section V presents the related work. Section VI concludes our work.

II. Approach

In this section, we present our approach in detail. As the test inputs of C compilers are C programs, in the remainder of this paper, we use the terms "test program" and "test input" exchangeably. Our approach consists of the following three steps.

- Our approach regards test programs as text and extracts tokens reflecting fault-relevant characteristics of the test program from text, and then transforms test programs into a set of vectors by counting the number of occurrences of each token for each test program.
- Our approach normalizes the values of elements in the vectors into an interval between 0 and 1.
- Our approach gives three prioritization strategies (i.e., greedy strategy, adaptive random strategy and search strategy) to prioritize the set of test programs.

In particular, in order to improve the prioritization efficiency, we propose to use the PCA processing to optimize our approach. An overview of our approach is shown in Fig. 1.

Section II-A presents the process of token extraction. Section II-B presents the process of normalization. Section II-C presents three prioritization strategies in detail. Section II-D presents the process of optimization (i.e., the PCA processing) in our approach.

A. Token Extraction

Our approach utilizes only test-input information to prioritize test programs. For C compilers, our approach first regards test programs as text. More concretely, our approach recognizes test programs as token streams. Then, our approach extracts tokens reflecting fault-relevant characteristics of the test program from text. More concretely, our approach considers three types of fault-relevant characteristics of the test program.

- Statement Characteristics: For C programs, "statement" is a kind of information reflecting fault-relevant characteristics of the test program. For example, if a test program does not have loop statements, the test program cannot detect loop optimization bugs. That is, as loop optimization bugs result from the existence of loop statements, loop statements characterize such bugs. In particular, statement keywords are the best way to extract statement characteristics from text. Therefore, our approach considers all statement keywords in C language as the first type of characteristics, e.g., for, while, if, else, goto, etc.
- Type and Modifier Characteristics: For C programs, "type" also reflects fault-relevant characteristics of the test program. For example, "struct" is a complex and error-prone variable type, and it usually leads to align bugs. Furthermore, modifier keywords are usually used together with type keywords in C language. Therefore, our approach regards type keywords and modifier keywords in C language as the second type of characteristics, e.g., struct, union, int, static, const, etc.
- Operator Characteristics: Test programs tend to contain many operators to implement some functions. In particular, when test programs contain a large number of operators and form a series of complex operations, the bugs related to operation optimization tend to be triggered. Therefore, our approach considers all operators in C language as the third type of characteristics, e.g., $++$, $--$, !, $||$, $>>$, etc.

After extracting the three types of tokens reflecting fault-relevant characteristics from text, our approach counts the number of occurrences of each token for each test program. Therefore, our approach transforms each test program into a text vector.

B. Normalization

As each element in a vector is numeric type, our approach normalizes each value of these elements in each vector into the interval $[0, 1]$ using the min-max normalization [23] in order to adjust values measured on different scales to a common scale. Supposed the set of test programs is denoted as $P = \{p_1, p_2, \ldots, p_n\}$ and the set of vectors representing P is denoted as $V = \{v_1, v_2, \ldots, v_n\}$, and the set of elements in

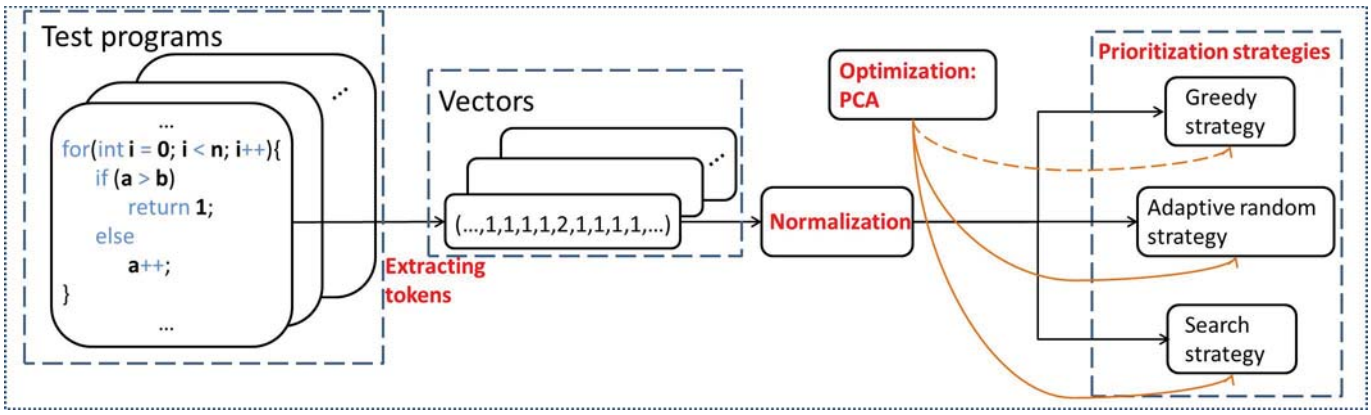


Fig. 1. Overview of Our Approach

a vector is denoted as $E = \{e_1, e_2, \ldots, e_m\}$, we use a variable x_{ij} to represent the value of the element e_j in the vector v_i before normalization and use a variable x^*_{ij} to represent the value of the element e_j in the vector v_i after normalization where $1 \leq i \leq n$ and $1 \leq j \leq m$. In particular, the normalization formula is shown as follows.

$$x^*_{ij} = \frac{x_{ij} - min(\{x_{kj} | 1 \leq k \leq n\})}{max(\{x_{kj} | 1 \leq k \leq n\}) - min(\{x_{kj} | 1 \leq k \leq n\})} \quad (1)$$

C. Prioritization Strategies

Based on the set of processed vectors, our approach gives three prioritization strategies to order the set of test programs, including greedy strategy, adaptive random strategy and search strategy.

1) Greedy Strategy: The traditional greedy strategy contains the total and additional techniques, both of which are based on coverage information [4]. However, our approach gets rid of coverage information and considers only test-input information, and thus we adapt the total technique to use in our approach. That is, the greedy strategy given by our approach is the adapted total technique.

Since each test program can be represented as a vector whose elements are numeric type and the values of them are normalized, our approach gives a score to each test program by calculating the Manhattan distance between the vector and origin vector $(0, 0, \ldots, 0)$. The formula calculating the score for each test program is expressed as follows.

$$score(v_i) = \sum_{k=1}^{m} |x_{ik}| \quad (2)$$

After giving a score to each test program, our approach prioritizes test programs in the descending order of their scores. That is, our approach uses the scores of test programs to replace coverage information in the traditional total technique to prioritize test programs.

2) Adaptive Random Strategy: Jiang et al. [24], [19] applied the idea of adaptive random testing to prioritize test cases. The adaptive random strategy selects next test case that has the maximum distance with the selected test cases. In particular, the set of unselected test cases is called the candidate set in adaptive random testing [25].

Similarly, our approach adapts the adaptive random strategy to prioritize test programs. In particular, our approach uses Manhattan distance to calculate the distance between two test programs. The formula calculating the distance between two test programs is expressed as follows.

$$distance(v_i, v_j) = \sum_{k=1}^{m} |x_{ik} - x_{jk}| \quad (3)$$

3) Search Strategy: Recently, Jiang and Chan [19] proposed to use the local beam search strategy to prioritize test cases, and achieved good effectiveness. Therefore, our approach adapts the search strategy (i.e., the local beam search strategy) to prioritize test programs based on the set of vectors.

In fact, the local beam search strategy can be regarded as the strategy between random order prioritization and the adaptive random strategy. That is, the adaptive random strategy calculates all the distances between selected test programs and the test programs in the candidate set for each selection, but the local beam search strategy samples randomly a subset of unselected test programs from the candidate set and calculates the distances between the subset of unselected test programs and all the selected test programs for each selection. In particular, our approach also uses Formula 3 to calculate the distance between test programs in this search strategy.

In particular, Manhattan distance is the default distance formula in our approach with the three strategies, but other distance formulae can be also used in our approach, which will be discussed in Section III-F4.

D. Optimization: Principal Components Analysis

Because the adaptive random and search strategies consider the interaction effect between test programs and the total time spent on calculating the distance between test programs tends

to be large especially when the number of test programs is large, which is confirmed in Section III-F1, our approaches with the two strategies need to be optimized. Therefore, we optimize our approaches with the two strategies by the PCA processing. The PCA processing improves the efficiency by reducing dimension for the set of normalized vectors before using the two strategies. Furthermore, for our approach with the greedy strategy, it does not include the interaction effect between test programs, and thus it does not have the high time cost issue. Therefore, our approach with the greedy strategy does not need to be optimized by the PCA processing[2].

PCA is a statistical method widely used in dimension reduction processing and it is based on the assumption that most information is contained in the directions along which the variations are the largest [26], [27]. Therefore, it uses an orthogonal transformation to convert a set of observations of possibly correlated elements into a set of values of linearly uncorrelated elements called principal components. These principal components retain the maximum possible variance of the original set. The number of principal components is smaller than or equal to the number of elements in the original vector. Therefore, the optimization by the PCA processing generates a set of new vectors that filter some elements including noise.

III. Experimental Study

In C compiler testing, the efficiency is as important as the effectiveness of a prioritization approach. Thus, we conduct an experimental study by applying our approach to C compilers, to investigate its efficiency as well as its effectiveness.

In particular, our study addresses the following four research questions. The first two research questions are concerned with the efficiency and effectiveness of our approach, and the other two research questions are concerned with the impact of various factors on our approach.

- RQ1: How does our approach perform comparing with the existing test case prioritization approaches in terms of efficiency?
- RQ2: How does our approach perform comparing with the existing test case prioritization approaches in terms of effectiveness?
- RQ3: How does the PCA processing impact our approach in terms of effectiveness and efficiency?
- RQ4: How do distance formulae impact our approach in terms of effectiveness and efficiency?

A. Subjects, Test Programs, and Bugs

We use two mainstream open-source C compilers in this experimental study, namely GCC[3] and LLVM[4] working in the x86_64-Linux platform. The two subjects cover all C compilers used in the literature of C compiler testing [20], [21], [22].

In particular, our study uses GCC 4.4.3 and LLVM 2.6, whose number of lines of code is 3,343,377 and 4,727,209, respectively.

Similar to the existing work on compiler testing [21], [28], [29], our study uses a random program generation tool CSmtih[5] [20] to generate a set of C programs, which serve as the test cases to be prioritized. In total, our study uses 82,593 test programs for each compiler.

Furthermore, we apply these test programs to each compiler and detect 20 bugs in GCC 4.4.3 and 14 bugs in LLVM 2.6. That is, the bugs used in our study are real compiler bugs. Note that such a number of bugs is not trivial according to the literature of compiler testing [20], [21]. Furthermore, Table I presents the number of test programs used to detect each bug, where the first and fourth rows represent the ID of each bug and the other rows represent the number of test programs detecting the corresponding bugs.

TABLE I
Density of Bugs

ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
GCC	1,220	1	1	2	1	1	7	1	3	1
LLVM	2	7	1	4	1	2	1	1	4	42
ID	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
GCC	1	3	1	23	7	2	7	28	1	1
LLVM	1	1	1	3	-	-	-	-	-	-

B. Independent Variables

We consider test case prioritization approaches as independent variables in the study.

1) Compared Approaches: In our study, we consider the following test case prioritization approaches as comparison approaches.

- Random order prioritization (RO): RO selects test programs randomly from the given test suite until all the test programs are selected. Furthermore, to reduce the influence of random selection on RO, for each test suite we repeat RO 10 times and use their average results. As for a test suite RO tends to produce various prioritization results, RO is not a practical prioritization approach. However, it usually serves as one of the baselines in the literature of test case prioritization [19], [30], [24] and so does it in this paper. In other words, RO actually represents the testing result without any prioritization approach.
- Input-based adaptive random prioritization (IARP): To our knowledge, IARP is the latest work on input-based test case prioritization [19], which prioritizes test programs based on the edit distance between selected test programs and unselected test programs.

[2]Dimension reduction by the PCA processing can optimize efficiency but it may also sacrifice effectiveness due to filtering some elements. That is, if an approach does not have efficiency issue, it does not need to be optimized by the PCA processing.

[3]http://gcc.gnu.org/.

[4]The LLVM project is a collection of compiler and toolchain techniques, which is accessible at http://llvm.org/. To be consistent with previous work on compiler testing, we also use LLVM to represent the compiler used in LLVM, which is mainly Clang.

[5]The C program generated by CSmith does not require external inputs, and its output is the checksum of the non-pointer global variables of the program at the end of the execution of the C program.

2) Our Prioritization Approach: As our approach gives various prioritization strategies (i.e., greedy strategy, adaptive random strategy and search strategy), it forms a family of text-vector based test case prioritization approaches. In particular, we denote our approach with the greedy strategy as TB-G, our approach with the adaptive random strategy as TB-AR and our approach with the search strategy as TB-S.

3) Variants of Our Approach: In order to explore the impact of various factors (i.e., the PCA processing and distance formulae) on our approach, we adapt our approach by changing these factors.

As the PCA processing is proposed to improve the efficiency of test case prioritization with the adaptive random strategy or the search strategy, we further implement two variants of our approach: (1) TB-AR$'$, our approach with the adaptive random strategy but without the PCA processing, and (2) TB-S$'$, our approach with the search strategy but without the PCA processing.

As distance formulae could affect the selection of test programs in test case prioritization, we implement five variants of our approach by using other distance formulae rather than Formula 2 or 3. In particular, the other two distance formulae are Euclidean distance and Cosine distance formulae, which are expressed by Formulae 4 and 5, respectively. For any two vectors $v_i = (x_{i1}, x_{i2}, \ldots, x_{im})$ and $v_j = (x_{j1}, x_{j2}, \ldots, x_{jm})$, Formula 4 calculates their Euclidean distance and Formula 5 calculates their Cosine distance. The Cosine distance is not implemented in our TB-G because Cosine distances requires the cosine value of the angle between two vectors, but the greedy strategy does not include the interaction effect between two test programs. We adopt Euclidean distance for TB-G by always setting one vector as $(0, 0, \ldots, 0)$. In summary, we implement the following variants of our approach: (1) TB-G with Euclidean distance, (2) TB-AR with Euclidean distance, (3) TB-AR with Cosine distance, (4) TB-S with Euclidean distance, and (5) TB-S with Cosine distance.

$$Euc(v_i, v_j) = \sqrt{\sum_{k=1}^{m}(x_{ik} - x_{jk})^2} \tag{4}$$

$$Cos(v_i, v_j) = 1 - \frac{\sum_{k=1}^{m}(x_{ik}x_{jk})}{\sqrt{\sum x_{ik}^2}\sqrt{\sum x_{jk}^2}} \tag{5}$$

C. Dependent Variables

The first dependent variable considered in our study is the time spent on prioritization, which is used to measure the efficiency of a test case prioritization approach.

The second dependent variable considered in our study is the number of executed test programs when detecting each bug. Based on it, we can calculate the APFD value, which is an effective measurement [4] on the rate of bug detection. APFD is the average rate of bugs detected during the execution of prioritized test programs and is widely used to measure the effectiveness of a test case prioritization approach. More concretely, supposed T is a test suite containing n test programs, T' is the prioritized test suite, and F is a set of m bugs detected by T, TF_i is the first test program in T' that detects the ith bug, we use Formula 6 to calculate the APFD value for T'.

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n} \tag{6}$$

D. Experimental Process

Initially, for the two C compilers, we randomly generate a large number of test programs using CSmith, which compose the test suite to prioritize. Then we apply these test programs to test each compiler, recoding the test programs that reveal each bug and the time spent on each test program execution.

First, we apply a family of our approaches with different prioritization strategies (i.e., TB-G, TB-AR and TB-S), the compared test case prioritization approaches (i.e., RO and IARP) to these test programs, recording the time spent on prioritization, the time spent on each step of our approach respectively, and the prioritization results, i.e., the execution order of the test programs. To reduce the impact of random selection on RO, we repeat RO 10 times and use the average results.

Second, in order to explore the impact of the PCA processing and distance formulae on our approach, we apply a family of variants of our approach including TB-AR$'$ and TB-S$'$ and the corresponding variants with different distance formulae to these test programs and record the same information as well. As the PCA processing targets at saving time on distance calculation, we further record the total time spent on distance calculation using our approach with PCA and our approach without PCA, respectively.

Finally, for each prioritization approach involved in our study, we calculate its APFD value.

All the experiments are conducted on a workstation with Intel E5504 Quad-Core Processor 2.0GHz and 100G memory, running Ubuntu 12.04.

E. Threats to Validity

The threats to internal validity mainly lie in the implementations of a family of our prioritization approaches with different prioritization strategies, the variants of our approach and the compared prioritization approaches. To avoid implementation error, at least two authors of this paper review the source code of these approaches. On the other hand, the implementations of these prioritization approaches could influence the efficiency of these approaches. To reduce this threat, we use C++ language to implement these approaches and try our best to optimize these implementations.

The threats to external validity mainly lie in the subjects and test programs. To reduce the threat of subjects, we use all the compilers that are used in the literature of C compiler testing [20], [21], but they may be not representative for other C compilers. To reduce the threat of test programs, we use a large number of C programs randomly generated by CSmith

as the prior work did [20], [21], but these C programs may not be representative for other C programs. In future, we will use more types of test programs (e.g., manually written C programs or test programs suited with compilers) so as to reduce this threat.

F. Results and Analysis

The following four subsections are to answer our four research questions, respectively.

1) Efficiency: In our study, we find that the time spent on prioritization using IARP is very long, even far longer than the time spent on test suite execution[6]. That is, IARP is infeasible for C compilers. The main reason for IARP's large prioritization cost is that IARP regards the whole test program as text and calculates the distance between them using edit distance. Besides, the time spent on prioritization using TB-S is also larger than the time spent on test suite execution, and thus TB-S is also infeasible for C compilers. Although extracting tokens to replace the whole text indeed saves a great deal of time, the time spent on prioritization using TB-S is still large, because the search strategy has heavy computational effort when the size of test suite is large (e.g., 82,593).

In fact, the time spent on our prioritization approaches can be divided into two parts. The first one (denoted as T_1) is the time spent on extracting tokens from text, transforming all the test programs into a set of vectors and processing the set of vectors, and the second one (denoted as T_2) is the time spent on ordering test programs using some prioritization strategy.

Fig. 2 presents the time spent on prioritization by the remaining feasible approaches—RO, TB-G and TB-AR. In this figure, the x axis represents the number of prioritized test programs, and the y axis represents the time spent on prioritization. When the value of x is 0, the value of y represents T_1 for each approach. In our study, T_1 of TB-G is 4,020 seconds, T_1 of TB-AR is 4,028 seconds, and T_1 of RO is 0 seconds. Furthermore, with the number of prioritized test programs increasing, the increment of T_2 of RO and TB-G is trivial. In our study, T_2 that RO takes to order all the test programs is 0.003 seconds, T_2 that TB-G takes to order all the test programs is 0.014 seconds, and T_2 that TB-AR takes to order all the test programs is 17,659 seconds.

From Fig. 2, the time spent on prioritization by TB-AR is much larger than that by RO and TB-G. With the number of test programs increasing from $50 * 10^3$ for TB-AR, the increment speed of time spent on prioritization slows down gradually. Moreover, although the time spent on prioritization by TB-AR is larger than that by RO and TB-G, it is still far less than the time spent on test suite execution because the time spent on prioritization by TB-AR is about six hours but the time spent on test suite execution is about three days.

In conclusion, the time spent on prioritization by TB-G and TB-AR is acceptable because it is far less than the time spent on test suite execution. TB-G is much more efficient than TB-AR. Besides, IARP and TB-S are infeasible since the time spent on prioritization is very long, even larger than the time spent on test suite execution, and thus we do not consider the two approaches in subsequent sections (i.e., Sections III-F2, III-F3 and III-F4).

[6]The time spent on test suite execution is about three days.

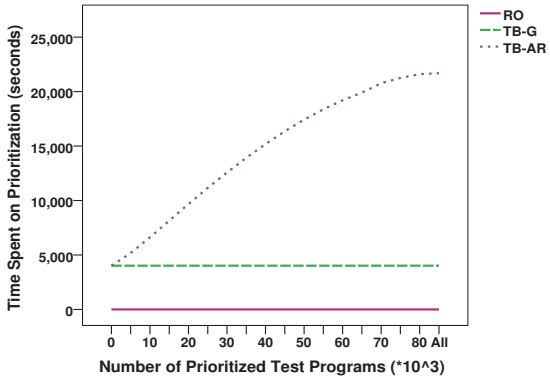
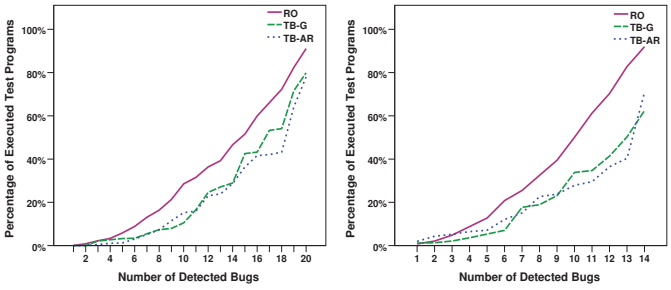


Fig. 2. Efficiency of Prioritization Approaches



(a) GCC (b) LLVM

Fig. 3. Percentage of Executed Test Programs to Reveal Each Bug

2) Effectiveness: Fig. 3 presents the percentage of test programs used to detect each bug through executing the prioritization results of RO, TB-G and TB-AR. In this figure, the x axis represents the number of detected bugs, and the y axis represents the percentage of executed test programs. In particular, Fig. 3(a) and Fig. 3(b) present the effectiveness of these approaches for GCC and LLVM, respectively. From the figure, the lines representing TB-G and TB-AR are almost always below the line representing RO. That is, when detecting the same number of bugs, the number of executed test programs by TB-G and TB-AR is smaller than that by RO, and thus TB-G and TB-AR are more effective than RO for both C compilers. Besides, the line representing TB-AR is below the line representing TB-G in most cases in Fig. 3(a), and thus TB-AR is more effective than TB-G for GCC; the line representing TB-AR is close to the line representing TB-G in Fig. 3(b), and thus the two approaches have close effectiveness for LLVM.

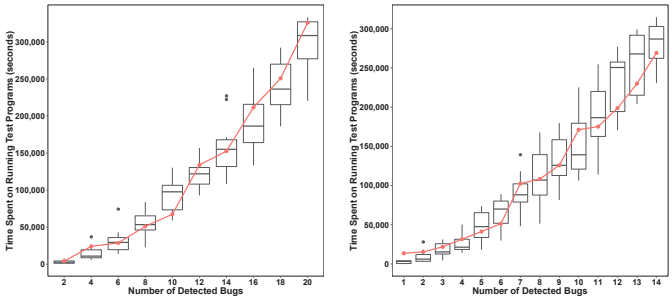
We further compare the AFPD values of the compared approaches, whose results are shown in Table II. From this table, for both GCC and LLVM the APFD values of TB-G and TB-AR are larger than that of RO, and thus TB-G and TB-AR are more effective than RO. For TB-G and TB-AR,

TABLE II
APFD VALUES OF PRIORITIZATION APPROACHES

Compiler	RO	TB-G	TB-AR
GCC	0.6611	0.7577	0.7790
LLVM	0.6400	0.7836	0.7829

TABLE III
ONE-TAILED PAIRED SAMPLE T TEST ($\alpha = 0.05$)

Compiler	RO v.s. TB-G	RO v.s. TB-AR
GCC	0.000026	0.000005
LLVM	0.000017	0.000017



(a) GCC (b) LLVM

Fig. 4. Test-Execution Time to Reveal Bugs

the APFD value of TB-AR is larger than that of TB-G for GCC, and their APFD values are close for LLVM, and thus TB-AR is slightly more effective than TB-G. Moreover, to learn whether TB-G and TB-AR significantly outperform RO, we further perform a one-tailed paired sample T test (with the significance level α set to be 0.05). The results are given in Table III. From this table, all the values are much smaller than 0.05, and thus TB-G and TB-AR are significantly more effective than RO in terms of APFD values.

Besides, Fig. 4 presents the total time (i.e., the time spent on prioritization and test execution) used to detect each bug by using RO and TB-G, where the y axis represents the total time spent on prioritization and test program execution before detecting the corresponding bug. Here we do not present the results of our proposed TB-AR because TB-G seems to be much more efficient but slightly less effective than TB-AR resulting from the preceding analysis. As RO has been repeated 10 times, we use boxplots to represent its results, and then we use the line to represent the results of TB-G. From this figure, the two prioritization approaches seem to achieve similar effectiveness in terms of time used to detect each bug. However, the proposed TB-G is more stable than RO, because the former approach is seldom higher than the higher quartile of RO (i.e., top of the box). That is, RO produces less stable prioritization results, whereas TB-G produces stable prioritization results although the latter seems to be less significantly effective. This observation is as expected, because similar to existing prioritization approaches, the proposed approach does not take into account test-execution time and assume the execution time of different test programs to be equal. Furthermore, this observation indicates an important practical aspect in test case prioritization, which should be addressed in future work.

In addition, we find that when the size of the test suite is small, the time spent on prioritization using TB-S is less than the time spent on test suite execution, and thus TB-S may be applicable for a small test suite. Therefore, in order to evaluate the effectiveness of TB-S, we conduct a study using a partial test suite (i.e., 30,000) from the entire test suite (i.e., 82,593), and prioritize it by various approaches (i.e., RO, TB-G, TB-AR, TB-S). Table IV shows their APFD values. From this table, TB-G and TB-AR are more effective than RO and TB-S for both two compilers, because the APFD values of TB-G and TB-AR are larger than those of RO and TB-S. That is, for TB-S, when the size of the test suite is small, it may be applicable in efficiency, but its effectiveness is worse than the effectiveness of TB-G and TB-AR. In particular, even when the size of the test suite is small, the time spent on prioritization using IARP is still larger than the time spent on test suite execution, and thus IARP is still infeasible.

TABLE IV
APFD VALUES FOR THE PARTIAL TEST SUITE

Compiler	RO	TB-G	TB-AR	TB-S
GCC	0.6594	0.7399	0.7283	0.6647
LLVM	0.6174	0.8429	0.8444	0.5314

In conclusion, TB-AR and TB-G are more effective than the other approaches in terms of APFD values, and TB-G is more stable than the other approaches in terms of time used to detect each bug. Besides, when the size of the test suite is small, TB-S may be applicable in efficiency but its effectiveness is worse than the effectiveness of TB-G and TB-AR.

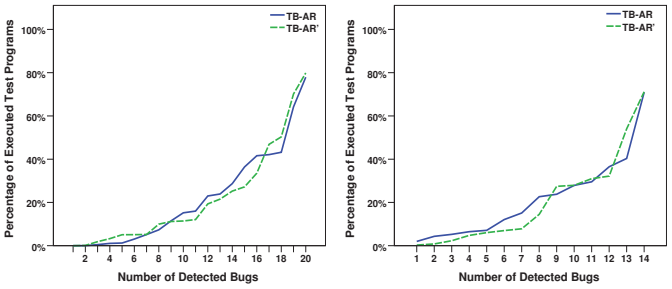
3) PCA Impact: For TB-AR and TB-S, our approach uses PCA to process vectors. Since TB-S is infeasible (explained in Section III-F1), we only evaluate the impact of the PCA processing on the effectiveness and efficiency of TB-AR. Similarly, when evaluating the impact of distance formulae in Section III-F4, we also do not consider TB-S.

Fig. 5 presents the impact of the PCA processing on the effectiveness of TB-AR. Table V presents the impact of the PCA processing on the APFD values and efficiency of TB-AR. In this table, the first two columns present the APFD values of TB-AR with the PCA processing (i.e., TB-AR) and TB-AR without the PCA processing (i.e., TB-AR$'$) on GCC and LLVM, respectively. The last column presents the time spent on the PCA processing and calculating all the distances between test programs (explained in Section III-D). The PCA processing targets saving time on distance calculation, but it may also incur extra cost from the process of PCA. Thus, in the last column of this table, the time before "+" represents the time spent on the PCA processing, and the time behind "+" represents the time spent on calculating all the distances between test programs.

From Fig. 5 and the first two columns in Table V, the effectiveness of TB-AR and TB-AR$'$ are close on both GCC

TABLE V
IMPACT OF THE PCA PROCESSING

Approach	APFD on GCC	APFD on LLVM	Time (seconds)
TB-AR	0.7790	0.7829	8 + 283
TB-AR′	0.7804	0.7948	0 + 1,599



(a) GCC (b) LLVM

Fig. 5. Effectiveness of the PCA Processing

and LLVM. That is, although the PCA processing reduces dimension of vectors, it has little impact on the effectiveness—only slight loss on effectiveness, which means that principle components indeed retain the important elements of vectors in our approach. From the last column in Table V, the total time spent on the PCA processing and calculating all the distances between test programs by TB-AR is much less than that by TB-AR′. That is, the PCA processing makes TB-AR more efficient due to dimension reduction.

In conclusion, the PCA processing improves the efficiency of TB-AR by slight loss on effectiveness.

4) Distance Impact: As distance formulae tend to impact the selection of test programs in test case prioritization, distance formulae are also an important factor that may have an impact on the effectiveness and efficiency of TB-G and TB-AR. In order to explore the impact of distance formulae, we introduce Euclidean distance and Cosine distance formulae to our approach besides Manhattan distance. In particular, we cannot use Cosine distance in TB-G (explained in Section III-B).

Fig. 6 shows the impact of distance formulae on the effectiveness of our approach. Besides, we also calculate their APFD values to assess their effectiveness, which is shown in Table VI. From this figure and this table, for TB-G, the effectiveness of Manhattan distance is close to that of Euclidean distance for both C compilers; for TB-AR, the effectiveness of the three distance formulae are close for GCC and the effectiveness of Manhattan distance seems to be better than that of Euclidean distance and Cosine distance for LLVM.

TABLE VI
APFD VALUES USING DIFFERENT DISTANCE FORMULAE

Sub.	TB-G		TB-AR		
	Manhattan	Euclidean	Manhattan	Euclidean	Cosine
GCC	0.7577	0.7661	0.7790	0.7765	0.7722
LLVM	0.7836	0.7947	0.7829	0.7674	0.7181

TABLE VII
TIME SPENT ON PRIORITIZATION USING DIFFERENT DISTANCE FORMULAE (SECONDS)

Approach	Manhattan	Euclidean	Cosine
TB-G	4,020	4,020	—
TB-AR	21,717	20,820	20,596

In addition, Table VII presents the time spent on prioritization by TB-G and TB-AR with different distance formulae. From this table, TB-G with different distance formulae has the same time spent on prioritization, and TB-AR with different distance formulae has similar time spent on prioritization. Therefore, distance formulae have no obvious impact on the efficiency of our approach.

In conclusion, our approach tends to be stably effective and efficient no matter which distance formula is used. Manhattan distance seems to be slightly more effective than the other two popular distance formulae.

5) Summary and Implication: In summary, we get the following findings.

- For C compilers, IARP and TB-S are infeasible due to their high time cost on prioritization.
- The time spent on prioritization using TB-G and TB-AR is acceptable. TB-G is much more efficient than TB-AR.
- TB-AR and TB-G are more effective than the other prioritization approaches in terms of APFD values, and TB-G is more stable than the other approaches in terms of time used to detect each bug.
- The PCA processing improves the efficiency of our approach by slight loss on effectiveness.
- Our approach tends to be stably effective and efficient no matter which distance formula is used. Manhattan distance is slightly more effective than the other two distance formulae.

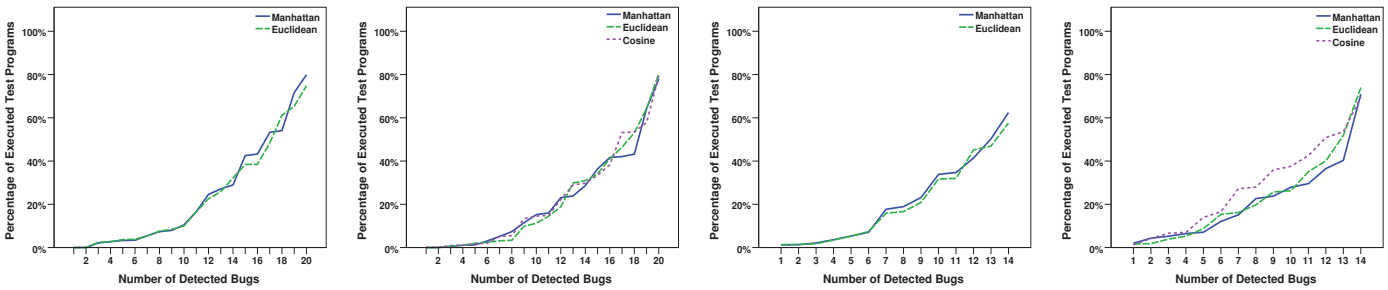
Furthermore, these findings imply that, because TB-G is much more efficient, and more stable in terms of time used to detect each bug, and slightly less effective than TB-AR in terms of APFD values, TB-G is more cost-effective than TB-AR for C compilers.

IV. DISCUSSION

A. Implicit Precondition of Test Case Prioritization

The goal of test case prioritization is to schedule the execution order of test cases so as to detect bugs as early as possible.

If the time spent on test suite execution is trivial, test case prioritization seems to be not so necessary, because it does not matter much whether a bug is detected one minute early or later. Therefore, only if the time spent on test suite execution is long, test case prioritization manifests its value, which is actually an implicit precondition of test case prioritization. However, most existing research [24], [31], [4], [19], [8] of test case prioritization evaluates their approaches in the scenario that the time spent on test suite execution ranges from only a few seconds to tens of minutes, thus they do not meet the



(a) TB-G for GCC (b) TB-AR for GCC (c) TB-G for LLVM (d) TB-AR for LLVM

Fig. 6. Impact of Distance Formulae

implicit precondition of test case prioritization. In our study, we use a real case—C compiler testing, and its time spent on test suite execution is very long. For example, Yang et al. [20] spend three years detecting 325 previously unknown C compiler bugs, and Le et al. [21] spend eleven months detecting 147 C compiler bugs. Thus, our study meets the implicit assumption of test case prioritization, and our results confirm the efficiency and effectiveness of our approach. Therefore, our work authentically manifests the real value of test case prioritization.

B. Extension of Our Approach

Our approach can be extended from two aspects: improving effectiveness and efficiency of our approach and expanding applications of our approach.

Our approach considers three types of fault-relevant characteristics of the test program including statement characteristics, type and modifier characteristics and operator characteristics. In fact, test programs have other important characteristics that may be related to bug detection, such as nest relations: if "struct" and "struct" are nested in a test program, it may lead to infinite loop on invalid struct redefinition. Furthermore, instead of representing a test program by a frequency vector, we may represent it by a sequence of tokens in the test program, which may help keep more accurate information of a test program. In future, we will improve the effectiveness of our approach by considering different representations of test programs.

In addition, in this paper, our approach targets at C compilers, but in fact, it can be easily extended to other compilers (e.g., Java compilers), even all the software systems whose test inputs are programs (e.g., operating systems and browsers). Since only the first step in our approach is specific to the software systems under test, our approach can be extended to different software systems by considering different tokens reflecting specific fault-relevant characteristics of the test input. For other compilers (e.g., Java compilers), the test inputs are also programs, and thus we could also extract similar tokens reflecting fault-relevant characteristics (e.g., statement characteristics, type and modifier characteristics and operator characteristics) like C compilers. For other software systems whose test inputs are programs/scripts (e.g., operating systems, image processing software, and browsers), we can also extract specific tokens reflecting fault-relevant characteristics by analyzing information related to bug detection. After extracting tokens reflecting fault-relevant characteristics from test-input text, we can apply the last two steps in our approach and then get the prioritized test suite. Therefore, our approach can be generalized from C compilers to the kind of software systems whose test inputs are programs.

V. Related Work

Since a lot of research focuses on test case prioritization, we classify them into four categories as prior work [8].

Prioritization Strategies. The research in this category mainly focuses on the strategies or algorithms used in test case prioritization. The most widely-used prioritization strategy is two greedy algorithms, namely total and additional algorithms [4]. However, as the two greedy algorithms cannot always achieve the best effectiveness [4], Zhang et al. [8] proposed models that unify the total and additional algorithms and generates a family of prioritization algorithms between them. Besides, researchers viewed test case prioritization as a search problem, and thus proposed various meta heuristics algorithms. Li et al. [32] proposed the 2-optimal strategy, which is a greedy algorithm based on the k-optimal algorithm [33]. In the same paper, Li et al. [32] also proposed a prioritization strategy based on hill-climbing algorithm and a prioritization strategy based on genetic programming algorithm. Dario et al. [34] proposed a hypervolume-based genetic algorithm for test case prioritization. Furthermore, Jiang et al. [24] proposed the adaptive random strategy using the idea of adaptive random testing [25], which selects the next test case that has the maximum distance with selected test cases. Recently, Jiang and Chan [19] proposed a prioritization strategy combining the adaptive random strategy and the search strategy (i.e., the local beam search algorithm). Tonella et al. [35] proposed a test case prioritization approach based on user knowledge using a machine learning algorithm. Yoo et al. [36] proposed a cluster-based test case prioritization approach based on their dynamic runtime behavior. Arafeen and Do [37] proposed a test case prioritization approach based on requirement clustering and traditional code analysis. Nguyen et al. [38] and Saha et al. [39] proposed to prioritize test cases based on information retrieval. In this paper, by referring to these existing

prioritization strategies, we adapt the greedy strategy (i.e., the total algorithm), adaptive random strategy and search strategy (i.e., the local beam search algorithm) by using test-input information and different distance formulae to prioritize test programs.

Criteria. The research in this category mainly focuses on various criteria used in test case prioritization. So far, most test case prioritization approaches utilize code-based coverage criteria including statement and branch coverage [4], block coverage [15], function coverage [5], modified condition/decision coverage [40], method coverage [15] and statically-estimated method coverage [17], [41]. Besides, Elbaum et al. [31] proposed to use the probability of exposing faults to prioritize test cases. Mei et al. [42] investigated dataflow coverage for testing service-oriented software. Staats et al. [43] used oracle coverage as a criterion. Korel et al. [44] used the coverage of system model instead of code-based coverage. Ma and Zhao [45] proposed to prioritize test cases by combining fault proneness and importance of modula. Xu and Ding [46] proposed to prioritize test cases based on transition coverage and roundtrip coverage. Bryce et al. [47] proposed to prioritize test cases based on window coverage and parameter coverage for event-driven software. Fang et al. [48] compared logic coverage criteria on test case prioritization. Furthermore, besides utilizing coverage criterion, Thomas et al. [18] and Jiang et al. [19] proposed to utilize test-input information to prioritize test cases. The former proposed to use the linguistic data (i.e., identifier names, comments and string literals) of test scripts to approximate their functionality and then gives high priority to test cases that test different functionalities, and the latter proposed to regard test inputs as text when test inputs are string and then prioritize them by calculating the distance between test cases using edit distance. Our approach also uses test inputs as the only input of test case prioritization, even similar to the approach of Thomas et al. [18] on the high level. However, as our approach transforms each test input (i.e., C program) into a text-vector representing fault-relevant characteristics of the test input, our approach is light-weight, and more effective than their work.

Constraints. The research in this category mainly focuses on various constraints of test case prioritization. Elbaum et al. [49] and Park et al. [50] investigated the constraints between test cost and fault severity. Hou et al [51] investigated the quota constraint on regression testing of service-centric systems. Kim and Porter [30] investigated the resource constraint, which may not allow the execution of the whole test suite. Walcott et al [52], Zhang et al. [7] and Do et al. [53] investigated time constraints, which need to select a subset of test cases for prioritization, by proposing a genetic algorithm, proposing an integer linear programming based technique and conducting an empirical study evaluating the effect of time constraints

Usage Scenarios. The research in this category mainly focuses on usage scenarios of test case prioritization. Test case prioritization is usually used in the regression scenario. In general, the regression scenario has two usage scenarios, including a specific subsequence version and a series of subsequent versions. Elbaum et al. [5], [31] referred to the first scenario as version-specific prioritization and the second one as general prioritization. Most research about test case prioritization focuses on approaches for general prioritization, but some researchers still investigated approaches for version-specific prioritization [54]. Furthermore, Elbaum et al. [5] investigated to use general prioritization approaches in version-specific prioritization. In particular, our approach not only can be used in the two scenarios in the regression scenario, but also can be used in the scenario where the test suite prioritized on the current version is used to test the current version.

VI. Conclusion

In this paper, we propose a novel text-vector based approach to prioritize test case for C compilers, which does not require any coverage information at all. In particular, our approach extracts tokens reflecting fault-relevant characteristics from program text, and transforms test programs into a set of text-vectors by counting the occurrence times of each token in each test program. After processing the set of vectors, our approach prioritizes test programs using three strategies (i.e., greedy strategy, adaptive random strategy and search strategy). To evaluate the efficiency and effectiveness of our approach, we conduct an experiment on two C compilers (i.e., GCC and LLVM). The experimental results show that our approach is much more efficient than the existing approaches and is effective in prioritizing test cases, and TB-G is more cost-effective than TB-AR and TB-S. Furthermore, we investigate the impact of various factors (i.e., the PCA processing and distance formulae) on the effectiveness and efficiency of our approach. The results show that the PCA processing improves the efficiency of our approach by slight loss on effectiveness, and distance formulae have no obvious impact on our approach.

VII. Acknowledgement

This work is supported by the National Basic Research Program of China (973) under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No. 61421091, 61432001, 61529201, 61522201, 61272089.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[2] Z. He, C. Liu, and H. Yan, "Software testing evolution process model and growth of software testing quality," *Science China Information Sciences*, vol. 58, no. 3, pp. 1–6, 2015.

[3] Y. Cheng, M. Wang, Y. Xiong, D. Hao, and L. Zhang, "Empirical evaluation of test coverage for functional programs," in *Proceedings of 9th International Conference on Software Testing, Verification and Validation*. IEEE, 2016, p. to appear.

[4] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings of the 1999 IEEE International Conference on Software Maintenance*. IEEE, 1999, pp. 179–188.

[5] S. 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*. ACM, 2000, pp. 102–112.

[6] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 10:1–10:31, 2014.

[7] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*. ACM, 2009, pp. 213–224.

[8] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 192–201.

[9] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, p. to appear, 2015.

[10] C.-T. Lin, C.-D. Chen, C.-S. Tsai, and G. M. Kapfhammer, "History-based test case prioritization with software version awareness," in *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems*. IEEE, 2013, pp. 171–172.

[11] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Proceedings of the 26th International Symposium on Software Reliability Engineering*. IEEE, 2015, p. to appear.

[12] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *Proceedings of 26th International Symposium on Software Reliability Engineering*. IEEE, 2015, pp. 46–57.

[13] Y. Lu, Y. Lou, S. Chen, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, p. to appear.

[14] D. Hao, X. Zhao, and L. Zhang, "Adaptive test-case prioritization guided by output inspection," in *Proceedings of the 37th Annual Computer Software and Applications Conference*. IEEE, 2013, pp. 169–179.

[15] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*. IEEE, 2004, pp. 113–124.

[16] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 302–311.

[17] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, 2012.

[18] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[19] B. Jiang and W. Chan, "Input-based adaptive randomized test case prioritization: A local beam search approach," *Journal of Systems and Software*, vol. 105, pp. 91–106, 2015.

[20] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011, pp. 283–294.

[21] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, pp. 216–226.

[22] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, p. to appear.

[23] Y. K. Jain and S. K. Bhandare, "Min max normalization based data perturbation method for privacy protection," *International Journal of Computer & Communication Technology*, vol. 2, no. 8, pp. 45–50, 2011.

[24] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *Proceedings of the 24th International Conference on Automated Software Engineering*. IEEE, 2009, pp. 233–244.

[25] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*. Springer, 2005, pp. 320–329.

[26] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1, pp. 37–52, 1987.

[27] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2002.

[28] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015, pp. 65–76.

[29] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 48, no. 6. ACM, 2013, pp. 197–208.

[30] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24rd International Conference on Software Engineering*. IEEE, 2002, pp. 119–129.

[31] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[32] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[33] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245–2269, 1965.

[34] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Hypervolume-based search for test case prioritization," in *Proceedings of the 7th International Symposium on Search-Based Software Engineering*. Springer, 2015, pp. 157–172.

[35] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. IEEE, 2006, pp. 123–133.

[36] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*. ACM, 2009, pp. 201–212.

[37] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 312–321.

[38] C. D. Nguyen, A. Marchetto, and P. Tonella, "Test case prioritization for audit testing of evolving web services using information retrieval techniques," in *Proceedings of the 9th IEEE International Conference on Web Services*. IEEE, 2011, pp. 636–643.

[39] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015, pp. 268–279.

[40] J. Jones, M. J. Harrold *et al.*, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[41] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing JUnit test cases in absence of coverage information," in *Proceedings of the 25th IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 19–28.

[42] L. Mei, Z. Zhang, W. Chan, and T. Tse, "Test case prioritization for regression testing of service-oriented business applications," in *Proceedings of the 18th International Conference on World Wide Web*. ACM, 2009, pp. 901–910.

[43] M. Staats, P. Loyola, and G. Rothermel, "Oracle-centric test case prioritization," in *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 311–320.

[44] B. Korel, L. H. Tahat, and M. Harman, "Test prioritization using system models," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE, 2005, pp. 559–568.

[45] Z. Ma and J. Zhao, "Test case prioritization based on analysis of program structure," in *Proceedings of the 15th Asia-Pacific Software Engineering Conference*. IEEE, 2008, pp. 471–478.

[46] D. Xu and J. Ding, "Prioritizing state-based aspect tests," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 265–274.

[47] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 48–64, 2011.

[48] C. Fang, Z. Chen, and B. Xu, "Comparing logic coverage criteria on test case prioritization," *Science China Information Sciences*, vol. 55, no. 12, pp. 2826–2840, 2012.

[49] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE, 2001, pp. 329–338.

[50] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement*. IEEE, 2008, pp. 39–46.

[51] S.-S. Hou, L. Zhang, T. Xie, and J.-S. Sun, "Quota-constrained test-case prioritization for regression testing of service-centric systems," in *Proceedings of the 24th International Conference on Software Maintenance*. IEEE, 2008, pp. 257–266.

[52] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ACM, 2006, pp. 1–12.

[53] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An empirical study of the effect of time constraints on the cost-benefits of regression testing," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2008, pp. 71–82.

[54] 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*. ACM, 2002, pp. 97–106.