Bypassing Code Coverage Approximation Limitations via Effective Input-Based Randomized Test Case Prioritization†

Bo Jiang  
School of Computer Science and Engineering  
Beihang University  
Beijing, China  
jiangbo@buaa.edu.cn

W.K. Chan‡  
Department of Computer Science  
City University of Hong Kong  
Tat Chee Avenue, Hong Kong  
wkchan@cityu.edu.hk

Abstract—Test case prioritization assigns the execution priorities of the test cases in a given test suite with the aim of achieving certain goals. Many existing test case prioritization techniques however assume the full-fledged availability of code coverage data, fault history, or test specification, which are seldom well-maintained in many software development projects. This paper proposes a novel family of LBS techniques. They make adaptive tree-based randomized explorations with an adaptive randomized candidate test set strategy to diversify the explorations among the branches of the exploration trees constructed by the test inputs in the test suite. They get rid of the assumption on the historical correlation of code coverage between program versions. Our techniques can be applied to programs with or without any previous versions, and hence are more general than many existing test case prioritization techniques. The empirical study on four popular UNIX utility benchmarks shows that, in terms of APFD, our LBS techniques can be as effective as some of the best code coverage-based greedy prioritization techniques ever proposed. We also show that they are significantly more efficient and scalable than the latter techniques.

Keywords—regression testing, adaptive test case prioritization

I. INTRODUCTION

Regression testing [10][11][12][15][17][18][23][25][30] assures programs not adversely affected by unintended modifications by running them over existing regression test suites. As such, intuitively, information collected in the previous executions of these test cases (e.g., the fault history, the change history, and execution profiles) on the previous versions can be used to optimize the current round of regression testing.

Many existing studies on test case reduction [26], selection [23] or prioritization [11] rely on the above assumption to formulate their proposed techniques or study the factors that may affect, say in the context of test case prioritization, the rate of coverage code [18] or the rate of fault detection achieved by these techniques [11].

† This research is supported in part by the Early Career Scheme of Research Grants Council of Hong Kong SAR (project number 123512) and the National Natural Science Foundation of China (project no. 61202077).

‡ Correspondence author

To what extent is this assumption valid (as researchers tend to like their work to be generally applicable)? We observe many counterexamples in practices, such as the problems of historic data unavailability (or partial historic data), the presence of newly defined test cases, and non-trivial implementation gaps between program versions. These issues significantly limit the applicability of techniques that rely on the above-mentioned assumption. For empirical studies on regression testing, these issues represent significant threats to internal and external validity. To the best of our knowledge, many empirical studies have not addressed these issues and have not adequately reported these issues in their studies.

Test case prioritization [12] is a widely studied aspect of regression testing techniques. It aims at permuting test cases in a given regression test suite without discarding any test case [18][30]. However, one test suite permutation may enable a failure-revealing test case to be executed by the program earlier than another permutation. Nonetheless, it is generally impractical to know in advance which test cases are failure-revealing.

Many existing heuristic [10][11][12][26][30] or meta-heuristics techniques [16][18] resolve to use the code coverage achieved by the regression test suites on previous versions of the program under test to guide these techniques to permute test suites.

To circumvent the difficulties in the use of code coverage data, a technique may use the black-box information of the test suites to guide necessary test case prioritization. However, formal program specifications are seldom available. Moreover, they may be inconsistent with the program version under test owing to poor maintenance. Program documentations in many software projects are also poorly maintained. Using a manual approach can be time-consuming. On the other hand, test inputs are generally available.

In this paper, we propose a family of novel input-based randomized local beam search (LBS) techniques for test case prioritization. They aim at spreading the sequences of test cases across the space modeled by the regression test suite as evenly as possible. Our LBS techniques can be regarded as a novel class of ART techniques for test case
prioritization [16]. Compared to the techniques proposed by Jiang et al. [16], our LBS techniques are able to work at the input level. Usually, the amount of information in execution profiling of a test case is significantly more than the amount of information in the input of the test case. Hence, it seems to be more challenging to effectively use the more limited amount of information for effective test case prioritization. Our approach is also able to work on more lightweight artifacts when testing many kinds of programs. Moreover, the work of Jiang et al. [16] only constructs one subsequence of test cases at any time; whereas, our LBS techniques make tree-based explorations over the input space rendered by each regression test suite. Intuitively, while all other factors being equal, our work is likely to be more effective than the work of Jiang et al. [16]. Finally, our proposed techniques only depend on input information, making them applicable to test program versions with or without any previous versions, and hence are more general than the techniques proposed by Jiang et al. [16]. When debugging a program, developers sometimes add new test cases or change a test case in the test harness, and execute the test suite again. While code coverage based techniques may be inapplicable as no coverage information is available for the changed test case, our approach is applicable to this scenario.

Our LBS techniques reduce the search space exploration cost (compared to full exploration) by controlling the number of branches in the exploration tree to be explored at any one round to a small number. Suppose that at each round, both the size of the candidate set and the number of branches to explore by our LBS techniques are $k$, and the number of distance comparisons between test cases in each node of the tree being explored is capped to be $m$, then there are at most $mk^2T$ comparisons, where $T$ is the size of the regression test suite.

The cost of our LBS techniques can be further reduced. If $T$ is a large number, one may further set both $m$ and $k$ to a small number. By so doing, the algorithm can be run in $O(Td)$ in worst case complexity, where $O(d)$ is the time complexity of the distance measure (e.g., string edit distance) chosen to initialize the algorithm. If both $m$ and $k$ are not controlled, finding the best distance will be an NP-hard problem. As such, a typical and well-known approach to deal with such a NP-hard search problem is to apply approximation algorithms. For brevity, we do not discuss this engineering component further in this paper.

We have evaluated our LBS prioritization techniques on four medium-sized UNIX utility programs to measure their cost-effectiveness in terms of APFD and the time spent on generating prioritized test suites. The empirical results are very interesting:

- The LBS techniques are significantly more effective than random ordering. The LBS techniques consistently score over 90%, whereas random ordering may only score 58–81%. The variants in APFD achieved by the LBS techniques are much smaller than these achieved by random ordering.
- The LBS techniques are significantly more effective than the code coverage-based techniques using the total strategy, including total-statement, total-branch, and total-function.
- The LBS techniques are not different in a statistically meaningful way from the two most effective coverage-based greedy prioritization techniques (namely, the additional-statement and additional-branch) ever proposed in the literature. The empirical result also shows that the LBS techniques are significantly more efficient and scalable than these two techniques.

The main contribution of the paper is as follows. (1) To the best of our knowledge, this paper is the first work that formulates a novel family of input-based adaptive random test case prioritization techniques. They do not suffer from the restriction on the code coverage availability on the previous versions of the program under test. (2) Also to the best of our knowledge, we have reported the first validation experiment using four UNIX utility benchmarks to validate input-based randomized test case prioritization techniques. The result shows that, in terms of APFD, our proposed input-based techniques can be highly effective and efficient to test medium-sized programs.

We organize the rest of paper as follows: we first review the preliminaries in Section II. Then, we describe our LBS prioritization techniques in Section III. After that, we present the empirical study that validates the LBS techniques in Section IV. Finally, we discuss the related work followed by the conclusion of the paper in Section V and Section VI, respectively.

II. PRELIMINARIES

A. Problem Formulation

Elbaum et al. [11] described the test case prioritization problem as follows:

Given: $T$, a test suite; $PT$, the set of permutations of $T$; $g$, a goal function from $PT$ to real numbers.

Problem: To find $T' \in PT$ such that $\forall T'' \in PT, g(T') \geq g(T'')$.

In the problem formulation, $PT$ represents the set of all possible prioritizations (orderings) of $T$ and $g$ is a goal function that calculates an award value for that ordering.

Test case prioritization can be viewed as a process that searches for a permutation of test suite to achieve this goal. For a test suite containing $N$ test cases, the search space is $N!$, which is intractable as the value of $N$ grows.
B. Problems of Code Coverage Based Regression Testing

The state of the art of code coverage based techniques still suffers from many limitations. In this section, we review five relatively important ones.

First, both fault history and execution profiles of the program over regression test cases are seldom well-maintained in real-world software development projects. For instance, in the repositories of much popular open-source software such as MySQL [36], Eclipse [33], and FireFox [34], no code coverage information is provided.

Some projects have well-maintained bug repositories. However, in many other cases, such as many Android applications (e.g., Foursquared [32]) available in Google Code [35], their corresponding bug repositories only record very few bug reports. Quite many of such bug reports have not been maintained as well.

One way to address them is to run a regression test suite on an older version of the program. However, the correctness criteria (e.g., the assertion statement in the JUnit test cases) may require manual determination. Both the executions of the test cases and the inspection and correction of correctness criteria of these test cases represent non-trivial overheads to the developers who follow the real-world practices to develop their programs.

Second, the program under regression test is a modified version of the same program. It is quite likely that some new test cases (for new features) have been added to the regression test suite. Nonetheless, it is relatively less likely that such a test case has been run on an older version of the program. For instance, an older version of the program may not be even possible to accept the new test case and run it.

Third, the coverage data on the present program version will be available only after testers have executed these test cases on the program version. However, if testers have already executed all the test cases on the present program version, it is pointless to prioritize the regression test suite because whether a test case has revealed a failure has been identified (and the most effective strategy is simply to assign those failure-revealing test cases to run prior to the other test cases). As a result, all these code coverage-based techniques inevitably use the code coverage data of the previous version of the same program to approximate the code coverage data achieved by the same test cases on the program version under regression test. This type of approximation is used heuristically. For instance, the code change between two versions of the same program can be very large. Take GNU flex [10] as an example. More than 12% (1731 lines) of the source code of this program have been changed from the version 2.5.1 to the version 2.5.2.

Recently, there are proposals to apply static analysis on the test harness with the programs version under regression test to collect an approximated coverage that may cover by each test case. It is a promising direction to remove the hurdle on the use of approximated code coverage on previous versions of the program under regression test.

Fourth, collecting the code coverage data requires profiling program executions, which can be impractical in many industrial settings. For example, in safety-critical software like avionics control that one of the authors of this paper can access, there is a lot of timing-related exception handling code. The profiling overhead may lead to unintended timeout exceptions. Similarly, in resource stringent scenarios like embedded and mobile applications, field-based testing (e.g., testing on the real devices) is the de facto approach, even though emulators are also used for some part of testing. However, if a regression test suite is to test the program as a whole, using emulators alone is inadequate. It is because program instrumentation may consume a lot of scarce memory and processing resources, which may interfere with the normal execution of the application under regression test, and yet the emulator of the embedded environment cannot fully emulate the features of the embedded platforms.

Last, but not the least, in cloud computing, (web) services normally and merely expose their interface information for other (web) services to select, invoke and compose. The source code of these services may not be accessible for code coverage profiling or (static and dynamic) analysis. It is very challenging to conduct effective third-party testing (e.g., through an independent service certification agency) and in-field regression testing.

C. Adaptive Random Testing (ART)

We recall that adaptive random testing (ART) [5] is a test case generation technique that makes use of the input information, spreading test cases as evenly as possible across the input domain. It aims at using a fewer amount of test cases to expose the first failure (in terms of F-measure) compared to the pure random testing (RT). If the extra time cost of ART is neither negligible [2][6] nor controllable, then RT may be able to reveal the first failure of each fault faster (in terms of the total amount of time spent so far) than ART [6].

Hence, the key issues in formulating a successful class of ART techniques include (1) controlling the extra time cost and (2) retaining a high effectiveness in a real-world development setting. The work of Jiang et al. [16] contribute to extend ART for test case generation to the domain of test case prioritization for regression testing, where the test cases have been generated and the size of a test suite is often limited. Both factors help to control the extra cost of ART mentioned above. Their techniques have shown to be highly effective.

Our paper further extends the contribution of Jiang et al. [16] to lower the cost of ART for test case prioritization so that ART can be applied to the input domain and still it is...
highly effective. We make this latter claim by our empirical study to be presented in Section IV. For instance, in the test case generation domain, ART is observed to be, on average, 11% more effective than RT. As we are going to present in the evaluation section of this paper, our LBS techniques consistently achieve more than 90% in terms of APFD, and random ordering varies from 58–81%. To the best of our knowledge, this work has ironed out the first set of effective input-based ART for test case prioritization.

III. RANDOMIZED LOCAL BEAM SEARCH TECHNIQUES

In this paper, we propose a family of novel input-based test case prioritization techniques.

A. Design Concerns on Candidate Set and Search Width

On the one hand, our techniques use the relative lightweight input information instead of code-coverage information for test case prioritization. They should be more efficient. On the other hand, because our generalized local beam search strategy keeps a beam width of \( k \) to search for the best partial solutions instead of merely one possibility at any one round during the search process, it incurs additional time cost because the search space is larger than simply searching for one solution each time as in the traditional approach to use the local beam search algorithm [4].

Thus, unlike the greedy techniques [11] that select the successors among all not yet prioritized test cases, our LBS techniques are designed to save time by randomly sampling a candidate set of \( c \) successors from all possible successors for each state as Jiang et al. [16] did.

Moreover, it only selects the best \( k \) (where \( k \leq c \)) successors of each state for further exploration. To simplify our presentation, we use \( k \) to refer to both the beam width and the number of successors expanded for each state. In general, it can be configured as two independent parameters.

Because the total number of successors (i.e., the number of remaining test cases to be prioritized) \( s \) must decrease while the subsequence of test case (i.e., a state of a local beam search) is being constructed, we use the minimum between \( s \) and \( c \) (denoted by \( \text{min}(s,c) \)) as the size of candidate set. Note that this strategy has not been developed by previous work [5][16]. Similarly, we use \( \text{min}(s,k) \) as the number of successors selected for each state (i.e., the number of branches at each expansion step).

Our techniques have also included a new design on the candidate set usage to make it more “diverse”. Specifically, each of our techniques prevents each unselected successor (i.e., test case) in a previous candidate set from including into the current candidate set until all test cases have entered the candidate set at least once. Then, it will ‘reset’ this inclusion label so that each remaining test case can be selected and included in a candidate set.

Our key insight on the above candidate set design is that the distribution of test cases in the inputs among the available regression test suites may not be even. Thus, test cases from a denser region will have higher chances to be selected into a candidate set than test cases from a sparier region.

To the best of our knowledge, all the ART techniques for test case generation do not have the above characteristics.

B. Measuring Even Spreading of Test Cases

We aim to allocate the regression test cases across the space rendered by a regression test suite as evenly as possible. We select from a successor pool those successors who are farthest away from those already prioritized test cases in each search step. Chen et al. [7] has shown that this simple strategy only ensure the test cases to be far away from each other, but cannot ensure the input domain to have the same density of test cases.

In statistics, Discrepancy measures whether different regions of a domain have equal density of sample points. The smaller the Discrepancy value, the more evenly spread are the set of test cases. Thus in our technique, we will use Discrepancy to select the final best \( k \) successors from the successor pool in each round. In this way, the selected successors can be both far away from each other and distributed within the input domain with equal density.

Discrepancy in [7] is as follows: Given an domain \( D \) and \( D_1, D_2, \ldots, D_m \) donate in rectangular sub-domains of \( D \) whose location and size are randomly defined; and given \( E \) represents the set of all test cases and \( E_i \) are the sets of test cases whose input is within domain \( D_i \) where \( 1 \leq i \leq m \).

\[
\text{Discrepancy}(E) = \max_{1 \leq i \leq m} \left\{ \frac{|E_i|}{|E|} - \frac{|D_i|}{|D|} \right\}
\]

A further generalization of discrepancy for a sequence is feasible and we leave it a future work.

C. The LBS Prioritization Algorithm

TABLE 1 show our LBS algorithm entitled prioritize. It accepts a set of unordered test cases \( T \), and generates a sequence of prioritized test cases \( P \). It first randomly selects \( k \) test cases from \( T \) as the initial \( k \) subsequences and puts them into \( S \) (line 4). Then it enters a loop to select one more test case in each round until all the test cases have been prioritized (lines 5 to 22).

In each round of iteration, the algorithm tries to find the best \( k \) successors by selecting one more test case (lines 6 to 21). It first determines \( c_r \), which stands for the size of the candidate set for \( S' \) (line 7), and randomly selects \( c_r \) test cases that have not been selected yet to form the candidate set \( C \) (line 8).

After that, it calculates the distance matrix \( D \) between test cases in the candidate set and the already-prioritized test
cases (line 9 to 12). Then it sorts the candidate test cases in descending order of their distance from the set of prioritized test cases (as defined by function \( f_2 \) (lines 13 to 16). Next, it selects the first \( k \) candidate test cases, appends each of them to \( S' \) to form \( k \) best successors, and puts them into the successor’s pool \( A \) (lines 17 to 19). When all the successors of \( S \) have been selected, it selects the best \( k \) successors from the pool \( A \) as the new \( S \) in the ascending order of their discrepancy values (line 21). Finally, if all the test cases have been prioritized, it selects a sequence of test cases with smallest discrepancy value from \( S \) (lines 23 and 24).

In this algorithm, the function \( f_2 \) determines the distance between the inputs of two test cases, which is best determined by the input structure and semantics of the application under test. For example, when the applications under test are numerical applications, we can use the Euclidean distance to measure the inputs. When testing command line applications like sed, we can use the string edit distance \( [14] \) to measure their distance.

\[
f_2(D, S', C_n) = \begin{cases} 
\min_{0 \leq m \leq |S'|} d_{mn} & (1) \text{ (see Chen et al. [5])} \\
\text{avg}_{0 \leq m \leq |S'|} d_{mn} & (2) \text{ (see Ciupa et al. [9])} \\
\max_{0 \leq m \leq |S'|} d_{mn} & (3) \text{ (see Jiang et al. [16])} 
\end{cases}
\]

In the algorithm, the function \( f_2 \) measures the distance between a candidate test case \( C_n \) and the set of already prioritized test cases \( S' \). There are three general strategies. We can use the minimum/average/maximum distance between the test case \( C_n \) and each test case in \( S' \) as shown by one of the equations (1)–(3) in the definition of \( f_2 \).

IV. EMPIRICAL STUDY

In this section, we report an empirical study that has evaluated the effectiveness of our LBS techniques.

A. Research Questions

We study two critical research questions.

RQ1: Are the input-based LBS techniques effective?

RQ2: Are the input-based LBS techniques efficient and scalable?

The answer to these questions will help to evaluate whether the proposed techniques can be more practical to be used in the testing of real-life programs that can be represented by our selected benchmarks.

B. Peer Techniques for Comparison

In our empirical study, we compared our LBS techniques with random ordering and six existing and highly effective coverage-based greedy prioritization techniques \([10][11][12]\) as shown in Table III. Due to the huge experimental effort, we will leave the comparison with the white-box ART techniques \([3]\) as future work.

| Algorithm: | Prioritize |
| Inputs: | \( T: \{t_1, t_2, \ldots\} \) is a set of unordered test cases |
| \( c \): Size of candidate set |
| \( k \): Beam width |
| Output: | \( P: \{p_1, p_2, \ldots\} \) is a sequence of prioritized test cases |
| \( S: |S'|1 \leq i \leq k \) is a set containing the best \( k \) successors (i.e., \( S' \) is subsequence of prioritized test cases) |
| \( A \): the pool of all the successors of the \( k \) best states |
| \( C \): is a set of candidate test cases |
| Randomly select \( k \) test cases from \( T \) as \( k \) initial subsequences into \( S \) |
| \( \text{while} |S'| \neq |T| \{ // \text{Goal state: all test cases are prioritized} 
| \text{foreach} (S in S') //Generate successor for each S' |
| \( c_i \) ~ \text{min}(|T'| - |S'|, c) |
| \( C \) ~ randomly select \( c_i \) test cases from \( T S' \) |
| \//calculate test case distance Matrix. |
| \( D: d_{ij}/c_i \) is a \( |S'| \times |C| \) dimensioned array |
| for \( m = 1, 2, \ldots, |S'| \) |
| for \( n = 1, 2, \ldots, |C| \) |
| \( d_{mn} \rightleftharpoons f_2(S'_{mn}) \) |
| \( V: \{v_n \mid 1 \leq n \leq |C| \} \) is a distance array represents the distance between \( C_n \) and the set of prioritized test cases. |
| for \( n = 1, 2, \ldots, |C| \) |
| \( v_n \rightleftharpoons f_2(D, S', C_n) \) |
| sort the test cases in \( C \) based on \( V \) descendingly |
| \( k_i \rightleftharpoons \text{min}(\{V|S_i\}, k) \) |
| \//select \( k_i \) best successor into the test pool |
| for \( n = 1, 2, \ldots, |k| \) |
| add \( S' \cup \{C_n\} \) to the pool \( A \) |
| \} \//foreach |
| \} \//while |
| \( P \rightleftharpoons \text{Select the sequence with lowest discrepancy from } A. \) |
| \( \} \//foreach |

<p>| Table II. Prioritization Techniques Evaluated |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>Random ordering</td>
<td>Randomly order test cases</td>
</tr>
<tr>
<td>Greedy</td>
<td>Code Coverage-based Strategy</td>
<td>Coverage Info. Used</td>
</tr>
<tr>
<td>total-st</td>
<td>Descending order of the total number of program constructs covered</td>
<td>statement</td>
</tr>
<tr>
<td>total-fn</td>
<td>Function</td>
<td>function</td>
</tr>
<tr>
<td>total-br</td>
<td>Branch</td>
<td>branch</td>
</tr>
<tr>
<td>addtl-st</td>
<td>Descending order of the coverage of program constructs not yet covered by the selected test cases.</td>
<td>statement</td>
</tr>
<tr>
<td>addtl-fn</td>
<td>Function</td>
<td>function</td>
</tr>
<tr>
<td>addtl-br</td>
<td>Branch</td>
<td>branch</td>
</tr>
<tr>
<td>LBS</td>
<td>Input-based Strategy</td>
<td>Test Set Distance ( f_2 )</td>
</tr>
<tr>
<td>LBS-maxmin</td>
<td>Equation (1)</td>
<td>Equation (1)</td>
</tr>
<tr>
<td>LBS-maxavg</td>
<td>Randomized local beam search</td>
<td>Equation (2)</td>
</tr>
<tr>
<td>LBS-maxmax</td>
<td>Equation (3)</td>
<td>Equation (3)</td>
</tr>
</tbody>
</table>
C. Subject Programs and Test Suites

We used four real-world UNIX utility programs (obtained from http://sir.unl.edu) in the evaluation. Table II shows their descriptive statistics.

Following [11], we used all the faulty versions whose faults can be revealed by more than 0% and less than 20% of the test cases.

D. Experimental Setup

Following [11], we generated 1000 test suites iteratively from the test pool such that each test suite can cover all branches in the program at least once. Since both the LBS techniques and random ordering are nondeterministic due to the impact of random seed, we repeated them 50 times. To reduce the huge computation cost in the experiment, we randomly selected 50 suites from all the available 1000 test suites for each of the UNIX programs. According to [1], 1000 runs per subject are sufficient for randomized techniques like our LBS techniques.

For the LBS techniques, we have evaluated different combinations of the candidate set size $c$ and beam width $k$ systematically with $c$ within the range {10, 20, 30, 40, 50} and $k$ in the range {3, 4, 5, ..., 20}, and the results of LBS techniques were generally consistent among different settings.

In total, we performed 40,000 prioritizations for each combination of $k$ and $c$ values for the LBS techniques. Due to space limitation, we report a representative case in this paper, which is $c = 10$ and $k = 7$, and it is also a balance of effectiveness and efficiency. We set the number of sub-domains $m$ as 500 when calculating the Discrepancy metric. Since our subject programs were all UNIX utility programs accepting command line and file input, we extracted the command line and file contents as strings, and use string edit distance [14] as function $f_i$.

We use APFD to measure the rate of fault detection [11]. APFD is the weighted average of the percentage of faults detected over the life of the suite. It is widely used in previous regression testing experiments: Let $T$ be a test suite containing $n$ test cases and let $F$ be a set of $m$ faults revealed by $T$. Let $TF_i$ be the first test case in the prioritized test suite $T'$ of $T$ that reveals fault $i$. The APFD value for $T'$ is given by the following equation:

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

We performed the experiment on a Dell PowerEdge 1950 server running a Solaris UNIX. The server has 2 Xeon 5355 (2.66Hz, 4 core) processors with 8GB physical memory.

E. Results and Analysis

1) Answering RQ1.

For each technique, we calculate the APFD results across all the faulty versions and draw box-and-whisker plots on each UNIX program as shown in Figure 1. The x-axis shows prioritization techniques and the y-axis shows APFD values.

From Figure 1, we can find that our LBS techniques perform well: they are better than both random and total greedy significantly on each program and even slightly better than the additional greedy techniques in general. The LBS techniques can achieve an APFD value of 0.95 on average, which is significantly better than random by more than 20% consistently on each program. Since the theoretical optimal prioritization result [12] is around 0.99, the LBS techniques are only 4% away from the optimal prioritization in terms of APFD values.

On gzip, flex and grep, the median values of LBS-maxavg and LBS-maxmin are slightly higher than those of all additional techniques. Since their notches have overlapping, the difference is not statistically meaningful at the 5% significance level. Moreover, the median values of LBS-maxmax on gzip, flex and grep are always not different in a statistically meaningful way from the additional techniques as their notches overlap. Finally, the LBS techniques perform especially well on sed: all LBS techniques perform significantly better than each additional technique in terms of medians.

We further perform both the ANOVA test and the multiple comparisons between each pair of techniques to see whether their means differ significantly.

Since all three input-based LBS prioritization techniques never overlap with the random and the total greedy techniques, we can further confirm the means of the input-based LBS techniques are significantly better than random and total greedy prioritization at the 5% significance level.

Among the LBS techniques, the LBS-maxavg and LBS-maxmin perform better. Specifically, LBS-maxavg is comparable to LBS-maxmin. Moreover, they both perform better than LBS-maxmax. This indicates maximizing the minimum or average distances is a better selection criterion to spread the test case sequence more evenly.

Finally, all LBS techniques are not different in a statistically meaningful way from the two best additional techniques (additional statement and additional branch). As shown by the two dotted blue lines in Figure 2, all LBS techniques perform better than additional function
This version is slightly different from the version published in *Proceedings of COMPSAC ’2013* (Research Track)

significantly. If we examine the figure carefully, we will find that both the LBS-maxmin and LBS-maxavg outperform the additional statement and additional function techniques slightly. However, the advantage is not statistically significant.

To answer RQ1: all our input-based LBS techniques are not different in a statistically meaningful way from the two best code coverage-based prioritization techniques (i.e., additional statement/branch).

2) **Answering RQ2**

In this section, we want to compare the input-based LBS techniques with the coverage-based greedy techniques in terms of efficiency. For each UNIX program, we randomly select 10%, 20%, 30%,...,100% of the test cases in the test pool as test suites, perform test case prioritization with random, total and additional greedy and input-based LBS prioritization techniques and record the prioritization time used.

As shown in Figure 3, the x-axis is the percentage of test pool used as test suite and the y-axis is the time used for test case prioritization averaged over all four UNIX programs. Since the prioritization times of different techniques vary a lot, we show the y-axis up to 20 seconds so the curve for most techniques can be shown clearly.

In general, we observe that the LBS techniques only use slightly more time than total function and random and are more efficient than other five greedy techniques. We see that the time spent by each of the additional (statement, function, and branch) coverage techniques grows exponentially when the size of test suite increases. For example, when using around 15% of the test pool, the prioritization time used by additional statement is more than 20 seconds. Moreover, the prioritization cost for additional statement will reach 700 seconds when using 100% of test pool (not shown in the figure). Except for the three additional techniques, the total statement and total branch spent more prioritization time than the input-based LBS techniques. The prioritization time used by different input-based LBS techniques almost overlaps with each other. Finally, the total function and random are more efficient than LBS techniques, but they are much less effective as already demonstrated in Figure 1.
Furthermore, the LBS techniques only grow with a small slope as test suite size increases. Therefore, we can answer RQ2 that the input-based LBS prioritization techniques are efficient and scalable techniques for real-life applications when compared with the coverage-based greedy techniques. Combined the answers to RQ1 and RQ2, we can conclude that the input-based LBS techniques can be as effective as the best code coverage-based techniques and yet are much more efficient and scalable.

![Figure 2. Multiple Comparison between Black-box LBS, Greedy and Random Techniques in terms of APFD](image)

F. Threats to Validity

In this study, we used several UNIX utility programs as subject programs. They are mainly medium-sized programs. Further empirical studies on other programs help further strengthen the external validity of our findings. Since our platform only supported C programs, our subject programs were all made of C programs. An empirical study on subject programs written in other popular programming languages will further strengthen our findings. For the UNIX programs, we use the branch-coverage test suites to conduct our empirical study. The way we generated branch-adequate test suite is the same as the procedure reported in [10]. In practice, there are other kinds of frequently used testing adequacy criteria. A further study on the test suites generated by other popular testing adequacy criteria will further strengthen the validity of our study.

Another factor affecting the threat to validity is the correctness of our tools. We used C++ to implement our framework for empirical study. To reduce bugs in our framework, we have carefully performed code inspection and testing on our tools to assure correctness.

In this work, we use the test pool provided by SIR, which has a fixed failure rate, to construct test suites. In practice, the failure rates of different test pools may be different, which may affect the comparison results and the APFD values obtained by each technique.

We have not compared Genetic Algorithm [15] for test case prioritization in this work. We will leave the full reporting of the comparison with generic algorithm techniques in future work.

V. RELATED WORK

In this section, we review the closely related work.

In [27], Yoo and Harman performed a comprehensive survey on test case minimization, selection and prioritization technique for regression testing and discussed open problems and potential directions for future research. Wong et al. [26] combined test suite minimization and prioritization techniques to select cases based on the cost per additional coverage. Walcott et al. [25] used genetic algorithms to reorder test cases under time constraints such that their techniques can be time-aware. Zhang et al. [30] used integer linear programming technique to find an optimal coverage solution, which can lead to effective test case prioritization results. Srivastava and Thiagarajan [24] proposed a binary matching technique to compute the changes between program versions at the basic block level and prioritize test cases to cover greedily the affected program changes.

Li et al. [18] empirically evaluated various search-based algorithms for test case prioritization in a systematic way. However, they concluded from their experiments that meta-heuristics approaches to optimizing the rate of coverage might not outperform the greedy algorithms. You et al. [30] evaluated time-aware test case prioritization on the Siemens suite and the program \textit{space}. They found that the differences among techniques in terms of APFD were not statistically significant. Li et al. [19] further showed that the Additional strategy and the 2-optimal greedy algorithms could be more effective than generic hill-climbing algorithms and the Total strategy. Qu et al. [22] proposed to group test cases according to their failure-exposing history and adjust their priority dynamically during executions. Although their techniques are black-box ones, they require execution history information that may not be available.
Jiang et al. [16] proposed code coverage based adaptive random test case prioritization. Compared to the result reported in [16], the result of our input-based LBS techniques is not different in a statistically meaningful way from their code coverage based techniques. Zhou [31] proposed to use coverage information to guide test case selection in ART, his work is similar to Jiang et al. [16] except that it uses the Manhattan distance to measure the distance between test cases.

Input-based or output-based test case prioritization is not completely new. Mei et al. [21] collected the interaction messages of a service, and used the tag information on these messages to prioritize a regression test suite for the service. Interaction messages may not be obtainable without prior execution of the service over the regression test suites. Our techniques have no this restriction. Zhai et al. [29] used the geo-location data in the inputs and the outputs of the test cases to assure location-based services. Our work does not make semantic assumption in the input data.

VI. CONCLUSION

Many existing test case prioritization techniques permute a test suite with the intention to improve the rate of fault detection via some other means. Almost all existing studies on prioritization techniques for regression testing use white-box code coverage information as such surrogate measures. These pieces of white-box information may be impractically to be obtained on the program under regression test in advance. Moreover, the coverage data on previous versions of the same program can be unavailable in many industrial projects. Data obtained from previous versions of the program may not be possible. Static analysis on the test harness with the source code of the program under test to get the approximated code coverage of individual test cases on the program under test is impriscise. In the computing cloud era, services only expose their interface information. The source code may not be accessible. Code coverage profiling or analysis is not a viable option for third-part testing (e.g., via an independent service certification agency). The use of code coverage for regression testing, through heuristically effective, can be impractically applied to address the industrial needs.

In this paper, we have developed a novel family of input-based test case prioritization techniques. The LBS techniques effectively prune unpromising search space and evenly spread the test cases within the space rendered by the inputs of the regression test suites. We have addressed the cost efficiency issue by having a novel design on the size of randomized candidate set and the depth of the local beam search. The validation experiment has shown that the input-based LBS techniques are not different in a statistically meaningful way from the two of the most effective code coverage-based techniques ever proposed in terms of APFD. At the same time, they have been shown to be significantly more efficient and scalable to deal with the test suite size requirements of medium-sized programs than the latter techniques. In terms of medium APFD, they also outperform random ordering by more than 20% on average, and achieve much smaller variants. Finally, the LBS techniques can be more general than many classical test case prioritization techniques because they are applicable test a program irrespective to whether the presence of previous versions of the program and whether the test information on previous versions have been available.

In future work, we will investigate further generalization of the LBS techniques and ART techniques for test case prioritization.

REFERENCES


