The Journal of Systems and Software 105 (2015) 91–106

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Input-based adaptive randomized test case prioritization: A local beam search approach[☆]

Bo Jiang[a], W.K. Chan[b,*]

[a] School of Computer Science and Engineering, Beihang University, Beijing, China
[b] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong

ARTICLE INFO

Article history:
Received 19 January 2014
Revised 1 March 2015
Accepted 22 March 2015
Available online 30 March 2015

Keywords:
Regression testing
Adaptive test case prioritization
Randomized algorithm

ABSTRACT

Test case prioritization assigns the execution priorities of the test cases in a given test suite. Many existing test case prioritization techniques assume the full-fledged availability of code coverage data, fault history, or test specification, which are seldom well-maintained in real-world software development projects. This paper proposes a novel family of input-based local-beam-search adaptive-randomized techniques. They make adaptive tree-based randomized explorations with a randomized candidate test set strategy to even out the search space explorations among the branches of the exploration trees constructed by the test inputs in the test suite. We report a validation experiment on a suite of four medium-size benchmarks. The results show that our techniques achieve either higher APFD values than or the same mean APFD values as the existing code-coverage-based greedy or search-based prioritization techniques, including Genetic, Greedy and ART, in both our controlled experiment and case study. Our techniques are also significantly more efficient than the Genetic and Greedy, but are less efficient than ART.

1. Introduction

Regression testing (Yoo and Harman, 2012) is a widely-practiced activity in real-world software development projects (Onoma et al., 1998), in which a better testing infrastructure has a potential to recover the economic loss resulting from software failures by one third (Tassey, 2002). During a session of a regression test, a changed program P is executed over a regression test suite T. Many companies executed the whole test suite to ensure the quality of their software (Onoma et al., 1998). Moreover, each nightly build of many open-source software projects such as *MySQL* (MySQL, 2013) and *FireFox* (FireFox, 2013) always apply the whole test suite to verify the version built.

If the time spent to complete the execution of a program over an individual test case is non-trivial, the time cost to execute the whole test suite T may be large (Jiang et al., 2011). For instance, profiling an execution trace of a C/C++ program at the memory access level using a *pintool* may easily incur tens to one hundred fold of slowdown (Luk et al., 2005). On the other hand, programmers may want to know the test results as early as possible at low cost.

Test case prioritization (Elbaum et al., 2002; Wong et al., 1997) is a *safe* aspect of regression testing. In essence, test case prioritization reorders the test cases in a test suite T and does not discard any test case in T for execution toward a chosen testing goal (denoted by G).

A vast majority of existing test case prioritization research studies (Yoo and Harman, 2012) propose to collect data from the executions of a previous version (denoted by Q) of P over a test suite T_{old} to guide the prioritization on T. For ease of presentation, we denote the set of program execution traces of Q over T_{old} by $Q(T_{\text{old}})$ and that of P over T by $P(T)$.

Numerous types of such data (such as the fault history (Kim and Porter, 2002), the change history (Elbaum et al., 2004), and the execution profiles (Elbaum et al., 2002)) obtained from these executions of Q have been empirically evaluated in diverse contexts with respect to the differences in their effects on regression testing results towards the selected goal G of regression testing techniques. For instance, a vast majority of empirical studies on test case prioritization validate on how quickly the permutations of T generated by such test case prioritization techniques detect faults in P by assuming that $T = T_{\text{old}}$. A recent trend is to replace the rate of fault detection by the rate of program element coverage (Li et al., 2007) or to incorporate the results of change impact analysis (Li et al., 2013) in their problem or solution

☆ This research is supported in part by the Early Career Scheme of Research Grants Council of Hong Kong SAR (project nos. 111313 and 11201114), the National Natural Science Foundation of China (project no. 61202077).

* Corresponding author. Tel.: +852 3442 9684.
E-mail addresses: jiangbo@buaa.edu.cn (B. Jiang), wkchan@cityu.edu.hk (W.K. Chan).

http://dx.doi.org/10.1016/j.jss.2015.03.066
0164-1212/© 2015 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

formulations. Still, the essential assumption of inferring T based on T_{old} remains unchanged.

In this paper, we propose a new family of novel input-based randomized local beam search (LBS) techniques for test case prioritization. This family of techniques targets to be applied in the general (more practical) scenario, where T may be different from T_{old} and Q may be different from P without taking any approximation (i.e., not assuming either T inferable from T_{old} or Q and P similar). Because both T_{old} and Q are irrelevant to this family of techniques, these LBS techniques can be applied in both regression testing and functional testing. In this way, practitioners need not care about whether a testing technique is applicable to functional testing scenarios only or to regression testing scenarios only, or both.

Given a test suite T, each LBS technique starts with a set of k partial solutions, each being a sequence of single test case taken from T. For each partial solution S, it randomly selects a number of test cases from $T \setminus S$ to construct a candidate set C and evaluates each extended partial solution S^{t}, where $t \in C$, according to a chosen distance measure. It marks the overall best k extended partial and randomized solutions as the current set X of partial solutions. It then goes into the next iteration to extend each partial solution in the current set X in the same manner until all test cases in T have been included in each partial solution X. It addresses the search space exploration cost problem by controlling the number of branches in the exploration tree in each round to a small number. Suppose that at each round, both the size of the candidate set and the number of branches to be explored by an LBS technique are k, and the number of distance comparisons between test cases in each node of the tree being explored is capped to be m, and then there are at most $mk^2|T|$ comparisons.

We have validated our LBS techniques on four medium-sized UNIX utility programs in a controlled experiment setting to evaluate their overall performance. We have further performed a case study on the comparison of our LBS search algorithm to the algorithms of Greedy (Elbaum et al., 2002), ART (Jiang et al., 2009), and Genetic (Li et al., 2007) by encoding test cases using input information and using the even-spread of test cases as the guidance heuristic to determine whether the performance of our techniques is merely attributed to the LBS algorithm. In both validations, we measured their effectiveness in terms of the average rate of fault detection (i.e., APFD (Elbaum et al., 2002)) and the time spent in generating a resultant prioritized test suite.

The empirical results from both the controlled experiment and the case study show that LBS achieves either higher mean APFD values than or similar mean APFD values as Greedy, ART, and GA. LBS also is *significantly* more efficient than GA but less efficient than ART at the 5% significance level. The result further shows that the effectiveness of LBS is not much affected by different parameter values needed to initialize the LBS algorithm. In the case study, we have the following observations: (1) the effectiveness of the studied LBS techniques was mainly contributed by our LBS algorithm, and (2) the use of input information for test case encoding and our heuristic also contributed to the significant reduction of the test case prioritization cost.

This work significantly extends its preliminary version (Jiang and Chan, 2013): (1) It generalizes the family of LBS techniques by presenting five more new techniques and evaluates the family against more existing techniques for benchmarking. (2) It reports a new experiment that investigates the impact of candidate set size and beam width on the effectiveness of the family. (3) It presents a new case study on comparing this family with several adapted classical search-based test case prioritization algorithms (Greedy, ART, and Genetic).

The main contribution of the paper together with its preliminary version (Jiang and Chan, 2013) is twofold. (1) This paper is the *first* work that presents a family of novel input-based randomized test case prioritization techniques. (2) It presents the *first* series of experiments to validate input-based search-based test case prioritization techniques.

We organize the rest of paper as follows: we review the preliminaries of this work in Section 2. Then, we describe our family of LBS techniques with their design rationales in Section 3. After that, we present validation experiments in Section 4 and Section 5. Section 6 discusses other issues relevant to our LBS techniques. Finally, we present the related work followed by the conclusion of the paper in Section 7 and Section 8, respectively.

2. Preliminaries

2.1. Problem formulation

Elbaum et al. (2002) 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 this problem formulation, PT represents a set of all possible prioritizations (orderings) of T and g is a goal function that calculates an award value for that ordering.

For a test suite containing N test cases, the size $|PT|$ is $N!$, which is intractable if N is large. In practice, the set PT in the universal quantification under the above problem formation is replaced by an enumerated subset of PT.

Moreover, a goal g, such as the rate of code coverage (Li et al., 2007), can be measurable before the execution of P over the prioritized test suite T'. Such a goal can be used by a search-based optimization technique for test case prioritization.

There are however other types of goals, such as the rate of fault detection (Elbaum et al., 2002), which cannot be measured directly before the execution of the test cases. A recent attempt is to use a heuristic (e.g., code coverage, even spreading of test cases) to make an approximation. Our techniques try to spread the test cases in T as evenly as possible within the input domain in each iteration using a randomized local beam search approach.

2.2. Critical review on assumptions of test case prioritization techniques

In this section, we revisit the assumptions made in the typical test case prioritization research work.

In general, T may be different from T_{old} and P may be different from Q. The dataset or execution profile of $Q(T_{\text{old}})$ is also unlikely to be the same as these of $Q(T)$, $P(T_{\text{old}})$, or $P(T)$. Moreover, if *either* a test case reduction/selection technique (Do et al., 2008; Yoo and Harman, 2012) *or* an impact analysis technique (Li et al., 2013) has been applied on T_{old} to construct a proper subset $T_{\text{old}'}$ of T_{old} and the test cases in $T_{\text{old}} \backslash T_{\text{old}'}$ have not been executed by Q, we have $Q(T_{\text{old}'}) \subset Q(T_{\text{old}})$. In this connection, if a test case prioritization technique relies on $Q(T_{\text{old}'})$ in prioritizing test cases in T, it is a threat.

We further categorize our observations on the limitations due to such assumptions into five classes:

First, assuming the historic data of T_{old} always available is restrictive. For instances, the execution profile data are seldom maintained in the repositories of real-world software development projects such as *MySQL* (MySQL, 2013), *Eclipse* (Eclipse, 2013), and *FireFox* (FireFox, 2013). In many projects, such as numerous Android applications (e.g., *Foursquared* (Foursquared, 2012)) available in Google Code (Google Code, 2013), their bug repositories only keep few bug reports.

One way to alleviate this problem is to run T on an older version Q. However, the correctness criteria (e.g., the assertion statements in JUnit test cases) may require manual determination. Both the executions of the test cases and the determination of their correctness lead to non-trivial overheads.

Collecting code coverage data requires profiling program executions, which may be impractical in some industrial environments. For instance, in safety-critical software like avionics control (where enumerators cannot support the execution of the whole test suite), there are many pieces of timing-related interrupt handling code. The profiling overhead may lead to unintended timeout exceptions.

Second, there are newly defined test cases in T (i.e., test cases in $T \backslash T_{\text{old}}$). Assuming that Q is able to execute each of such test cases is unrealistic as well. The program P is a changed version of Q. It is quite likely that some new test cases (for new features) have been added to the regression test suite. Nonetheless, it is unlikely that such a test case has been run over the older version Q of the program.

Third, there are revisions of test cases that are common to both T and T_{old}. Testers need to map between these test cases, which in general is a many-to-many mapping.

Fourth, the implementation gap between P and Q can be non-trivial. Take GNU *flex* (Flex, 2013) as an example. More than 12% (1731 lines) of the source code of *flex* has been changed from version 2.5.1 to version 2.5.2, and we found that many of the test cases applicable to both versions have changed their coverage statistics between the two versions.

Last, in cloud computing, web services merely expose their interface information and keep their implementation (including source code) inaccessible. Conducting third party testing (e.g., through an independent service certification agency) or in-field regression testing is challenging.

The former two classes are due to the unavailability of a prior execution data. The next two classes make the execution data extracted from $Q(T_{\text{old}'})$ or $Q(T_{\text{old}})$ unsound for the regression testing on P. The fifth class motivates us further to exploit input information to prioritize test cases.

As such, suppose that a controlled experiment on a test case prioritization technique M only evaluates the extent of M affected in the scenarios of either $T = T_{\text{old}} \wedge Q = P$ or its close approximation (i.e., $T \approx T_{\text{old}} \wedge T \subseteq T_{\text{old}} \wedge Q \approx P$, for some similarity judgment criterion $\approx$). Such a controlled experiment incurs significant threats to internal and external validities because the crucial element of realism in the study has been significantly compromised.

To circumvent the limitations associated with using the execution profile data extracted from $Q(T_{\text{old}})$, a prioritization technique may use the static information (data) associated with the program P or the test suite T to guide the prioritization process. Such static information can be either static program specification or information gained from static analysis. Precise program specifications of P are rarely available in real-world projects. Besides, using such information requires an accurate mapping from individual test cases to such information, which may not be sound.

Another way to alleviate the problem is to apply static analysis on P. Since static analysis does not involve any actual execution of the program, they often only provide conservative over-approximations of all possible execution traces of each test case. Nonetheless, if both $|T|$ and P are large in scale or involve concurrent components, static analysis techniques often fail to scale up to identify precise test case coverage information. Applying static analysis techniques to regression testing however still has a clear merit because $P(T)$ is only conservatively approximated by its own static version instead of approximated by Q or $Q(T_{\text{old}})$. If we sacrifice precision for efficiency, test case prioritization may be misled by the false positive issues in the approximated coverage data.

Impact analysis (Lahiri et al., 2010; Li et al., 2013; Rovegard et al., 2008) has a potential to precisely identify the change in between P and Q. Such impact analyses may be safe under certain conditions (Rothermel and Harrold, 1997). Adding a change impact analysis to figure out the difference between P and Q is still unable to eliminate the discrepancy between their corresponding execution profiles.

The test suite T alone contains much information. Suppose that the details (including the comments) of T are well-maintained. In this case, linguistic data from each test case (e.g., comment, string literal) in T can be extracted to guide the prioritization process that maximizes the average distance from the already-prioritized test cases (Thomas et al., 2014). The distribution of real world coordinates of points-of-interest in the *expected outputs* and inputs of test cases in T have been proposed to prioritize test cases in T (Zhai et al., 2014). It does not rely on the mapping information between the test case and program version. However, such techniques are inapplicable to prioritize non-coordinate based test cases.

Random ordering (Arcuri et al., 2012; Elbaum et al., 2002) is a strategy that can perfectly be applied in the general scenario, but is usually ineffective.

Adapting existing code-coverage-based techniques to use test input information requires a new formulation of what to be covered and the notion of coverage equivalence. These techniques also require evaluations to validate their effectiveness. This paper contributes to all of these aspects.

Manual approaches to test case prioritization are flexible but tedious to repeat precisely. Some test cases may be selected and run first based on their associated *risks*, which are often a managerial aspect of test case prioritization. For brevity, we do not discuss them further in this paper.

2.3. Review on algorithms for test case prioritization

This section revisits the test case prioritization techniques evaluated in our controlled experiment and case study.

2.3.1. Total and additional greedy algorithms

The Greedy algorithms studied in our experiment include the *total statement* technique and the *additional statement* technique (Elbaum et al., 2002). The total statement (total-st) test case prioritization technique sorts test cases in descending order of the total number of statements covered by each test case. In the case of a tie, it selects the involved test cases randomly. The additional statement (addtl-st) prioritization technique selects, in turn, the next test case that covers the maximum number of statements not yet covered in the previous round. When no remaining test case can improve statement coverage, the technique will reset all the statements to “not yet covered” and reapply addtl-st on the remaining test cases. When more than one test case covers the same number of statements not yet covered, it selects one of them randomly.

2.3.2. 2-Optimal algorithm

The 2-Optimal algorithm (2-Opti) (Lin, 1965; Skiena, 1998) is another typical greedy algorithm for test case prioritization. Similar to the additional greedy algorithm, it iteratively selects test case(s) to maximally cover those not yet covered statements in the previous round. Different from additional greedy, it selects two best test cases maximizing additional coverage rather than one in each round. When the complete coverage has been achieved, 2-Opti resets the coverage.

2.3.3. Hill climbing

We evaluated the steepest ascent hill climbing algorithm (Hill) (Li et al., 2007) for test case prioritization. The algorithm first randomly constructs a test suite permutation to make its current state. Then it evaluates neighbors of the current state. A neighbor of the current state is another test suite permutation constructed by exchanging the position of the first test case with any test case in the current permutation. It then moves to the neighbor with largest increase in fitness from the current state (no move if no neighbor has a larger fitness value). Then, the above movement is repeated until there is no state with higher fitness value to go to. The test suite permutation of the final state is returned as the prioritization result. In our experiment, we follow the algorithm presented in (Li et al., 2007) to implement the technique.

2.3.4. Genetic algorithm

Genetic Algorithm (GA) (Holland, 1975) is a class of adaptive search techniques based on the processes of natural genetic selection. It first randomly generates a set of permutation as the initial population. Then individual permutation of the initial population is evaluated. Finally, pairs of selected individuals are combined and mutated to generate new permutations as the new population. The process repeats until the maximally defined number of iteration is reached.

The implementation of the GA can vary a lot, depending on the choice of test suite encoding, fitness function, crossover strategy, mutation strategy, population size, and termination condition. In the evaluation of this work, we follow the implementation consideration presented in (Li et al., 2007). Specifically, we use the position of test cases in the permutation as the encoding. The fitness function is the Baker's linear ranking algorithm. The crossover strategy is to divide the parent encoding into two parts at a random position, and then exchange the two parts of two parent encodings to generate two offspring encoding. The mutation strategy used is to exchange two positions randomly within the sequence. Finally, the population size, the termination condition (i.e., the number of generations), the crossover probability, and the mutation probability are set as 100, 100, 0.8, and 0.1, respectively.

3. Our adaptive randomized local beam search techniques

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

3.1. Design concerns on candidate set and search width

On 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.

Thus, unlike the greedy techniques (Elbaum et al., 2002) that select the successors among all not yet covered 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. Jiang et al. (2011) 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 min(s,c)) as the size of candidate set. Note that this strategy has not been developed by previous work (Carlson et al., 2011; Jiang et al., 2011). Similarly, we use 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 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 enough. Thus, test cases from a denser region will have higher chances to be selected into a candidate set than test cases from a sparser region. To the best of our knowledge, no adaptive random testing techniques for test case generation and test case prioritization have the above characteristics.

3.2. 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. (2004) has shown that this simple strategy only ensures 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 techniques, we 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 faraway from each other and distributed within the input domain with equal density.

Discrepancy in Chen et al. (2007) is defined as follows: given domain D and $D_1, D_2, \ldots, D_m$ donate m 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 $1 \leq i \leq m$. A further generalization of discrepancy for a sequence is feasible and we leave it as a future work.

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

3.3. The LBS prioritization algorithm

Table 1 shows 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 an ordered set S (line 4). Then it

Table 1
Randomized local beam search algorithm.

```
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:     $P$: $\langle p_1, p_2, \ldots\rangle$ is a sequence of prioritized test cases
1   $S$: $\{ S^i \mid 1 \leqslant I \leqslant k \}$ is a set containing the best $k$ successors
    (i.e., $S^i$ is subsequence of prioritized test cases)
2   $A$ is the pool of all the successors of the $k$ best states
3   $C$ is a set of candidate test cases
4   Randomly select $k$ test cases from $T$ as $k$ initial subsequences
    into $S$
5   while($|S^1|$ != $|T|$){ //Goal state: all test cases are prioritized
6     foreach ( $S^i$ in $S$ ){ //Generate successor for each $S^i$
7       $c_i \leftarrow \min(|T|-|S^i|, c)$
8       $C \leftarrow$ randomly select $c_i$ test cases from $T \backslash S^i$
        //calculate test case distance Matrix.
9       $D$: $d_{|S^i||C|}$ is a $|S^i| \times |C|$ dimensioned array
10      for $m = 1, 2, \ldots, |S^i|$
11        for $n = 1, 2, \ldots, |C|$
12          $d_{m,n} \leftarrow f_1(S^i_m, C_n)$
13      $V$: $\{ v_n \mid 1 \leqslant n \leqslant |C| \}$ is a distance array represents the
    distance between $C_n$ and the set of prioritized test cases.
14      for $n = 1, 2, \ldots, |C|$
15        $v_n \leftarrow f_2(D, S^i, C_n)$
16      sort the test cases in $C$ based on $V$ descendingly
17      $k_i \leftarrow \min(|T|-|S_i|, k)$
        //select $k_i$ best successor into the test pool
18      for $n = 1, 2, \ldots, |k_i|$
19        add $S^i \cup \{C_n\}$ to the pool $A$
20    }//foreach
21    $S \leftarrow$ Select the $k$ best successors with lowest discrepancy
    from $A$.
22  }//while
23  $P \leftarrow$ Select the sequence with lowest discrepancy from $S$
24  return $P$
```

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_i, which stands for the size of the candidate set for S^i (line 7), and randomly selects c_i 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 (lines 9–12). Then it sorts the candidate test cases in descending order of their distances from the set of prioritized test cases as defined by function f_2 (lines 13–16). Next, it selects the first k_i candidate test cases, appends each of them to S^i to form k_i best successors, and puts them into the successor's pool A (lines 17–19). When all the successors of S have been selected, it selects the best k successors from the pool A as the new S in 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 the smallest discrepancy value from S (lines 23 and 24).

In this algorithm, the function f_1 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* (Gusfield, 1997) to measure their distance.

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^i. In this work, we extend its conference version to propose the family of generalized LBS techniques by extending the definition of f_2, which we will discuss in the following section.

$$f_2(D,\ S^i, C_n) = \begin{cases} \min\limits_{0\le m\le |S^i|} d_{mn} \quad (1) \ \text{(see Chen et al. [5])},\ \text{LBS}_{100} \\ \operatorname{avg}\limits_{0\le m\le |S^i|} d_{mn} \quad (2)\ \text{(see Ciupa et al. [9])},\ \text{LBS}_{010} \\ \max\limits_{0\le m\le |S^i|} d_{mn} \quad (3) \ \text{(see Jiang et at. [16])},\ \text{LBS}_{001} \\ \text{random}\left(\min\limits_{0\le m\le |S^i|} d_{mn},\ \operatorname{avg}\limits_{0\le m\le |S^i|} d_{mn}\right) (4),\ \text{LBS}_{110} \\ \text{random}\left(\min\limits_{0\le m\le |S^i|} d_{mn},\ \max\limits_{0\le m\le |S^i|} d_{mn}\right) (5),\ \text{LBS}_{101} \\ \text{random}\left(\operatorname{avg}\limits_{0\le m\le |S^i|} d_{mn},\ \max\limits_{0\le m\le |S^i|} d_{mn}\right) (6),\ \text{LBS}_{011} \\ \text{random}\left(\min\limits_{0\le m\le |S^i|} d_{mn},\ \operatorname{avg}\limits_{0\le m\le |S^i|} d_{mn},\ \max\limits_{0\le m\le |S^i|} d_{mn}\right) (7),\ \text{LBS}_{111} \end{cases}$$

3.4. Generalization of LBS prioritization algorithm

The definition of test set distance is shown in the equation below. As mentioned in the last section, we generalize the LBS techniques by extending the definition of test set distance f_2 in the conference version. In the conference version, f_2 is defined as the minimum (Eq. (1)) or average (Eq. (2)) or maximum (Eq. (3)) distances between the candidate test case and the already selected test cases across all beam steps. However, fixing the test set distance to one definition across all beam steps may also limit the evenly spread of the test cases. The most flexible test set distance definition is to randomly select one of the three test set distances in each beam step.

Suppose that we use the encoding ⟨*min, avg, max*⟩ to represent the test set distance in a beam step, where "*min*", "*avg*", or "*max*" is a Boolean variable. A true value (i.e., 1) means the corresponding test set distance can be selected in the beam step. Similarly, a false value (i.e., 0) means that the corresponding test set distance cannot be selected in the beam step. Thus, the strategy of randomly selecting one of the three test set distances in a beam step can be encoded as ⟨1,1,1⟩ (which forms the technique LBS_{111} using Eq. (7)). Similarly, randomly selecting one test set distance among "*avg*" and "*max*" only is encoded as ⟨0,1,1⟩ (which forms the technique LBS_{011} using Eq. (6)). The other LBS techniques shown in the equations can be interpreted similarly. So, in general, there are seven LBS techniques: LBS_{100}, LBS_{010}, LBS_{001}, LBS_{110}, LBS_{101}, LBS_{011}, and LBS_{111}.

3.5. Relationships with adaptive random test case prioritization (ART)

The most closely related work of LBS is the adaptive random test case prioritization (ART) (Jiang et al., 2009). We recall that adaptive random testing (Chen et al., 2004) 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 adaptive random testing for test case generation is neither negligible (Arcuri and Briand, 2011; Chen et al., 2004) 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 adaptive random testing for test case generation (Chen et al., 2004).

Hence, some of 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. (Jiang et al., 2011) contributes to extend adaptive random testing 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 effective. The prioritization costs of their techniques are significantly lower than that of the additional series of test case prioritization techniques. However, shared with other code coverage-based techniques, these techniques require execution profiling, which incurs additional prioritization costs.

LBS can be regarded as a novel class of ART (Jiang et al., 2011). Our paper contributes to lower the cost of ART for test case prioritization to prioritize test cases based on the information in the test inputs of the test cases, and still LBS is highly effective. Usually, the amount of data from execution profile of a test case is significantly more than the amount of data in the input of the same test case. Hence, it is more challenging for LBS to effectively use the more limited amount of data for effective test case prioritization. We make this claim by our controlled experiment to be presented in Section 4. For instance, in the test case generation domain, adaptive random testing is observed to be, on average, 11% more effective than RT. As we are going to present in the evaluation section of this paper, the LBS techniques consistently achieve higher than 90% in terms of APFD, and random ordering varies from 58 to 81%.

To the best of our knowledge, this work has ironed out the first set of effective input-based ART for test case prioritization. We are also unaware of existing works that have applied other search-based algorithms to input-based test case prioritization.

Moreover, ART only constructs one subsequence of test cases at any time, whereas, LBS makes a tree-based search over the input space rendered by each regression test suite. Intuitively, while all other factors equal, LBS is likely to be more effective but less efficient than ART in exploring the solution space.

4. Controlled experiment

In this section, we report a controlled experiment evaluating the effectiveness and efficiency of the LBS techniques.

4.1. Research questions

We study three critical research questions as follows:

RQ1: Is the family of input-based LBS techniques effective?

RQ2: Is the family of input-based LBS techniques efficient?

RQ3: Will the size of the candidate set and beam width impact on the effectiveness of input-based LBS techniques?

The answer to RQ1 clarifies whether the input-based LBS techniques can increase the rate of fault detection in regression testing. Furthermore, it enables us to examine whether LBS techniques using an atomic test set distance (i.e., LBS_{100}, LBS_{010}, and LBS_{001}) share the same effectiveness with the LBS techniques using a composite of two test set distance measures (i.e., LBS_{110}, LBS_{011}, and LBS_{101}) or more (LBS_{111}). The answer to RQ1 will also tell us the relative effectiveness of LBS with respect to classical code-coverage-based search-based techniques for test case prioritization.

The answer to RQ2 validates whether the prioritization cost of LBS techniques is low. It also reveals whether LBS techniques using composite test set distances may be different significantly from LBS techniques using atomic test set distance in terms of efficiency. The study will also tell us about the relative efficiency of LBS techniques as compared with classical code-coverage-based search-based algorithms.

The answer to RQ3 explores the design space and guides the testers on the direction on selecting a more appropriate candidate set size and beam width.

4.2. Experimental setup

In this section, we present the setup of our experiment.

4.2.1. Control variables

Our **platform** to conduct our experiment was a Dell PowerEdge M520 server running Ubuntu Linux. The server was configured with a Xeon E5-2400 (16 cores) processor with 12GB physical memory.

For **Subject Programs** and **Test Suites**, we used a suite of four real-world UNIX benchmarks (downloaded from http://sir.unl.edu) in our evaluation. Table 2 shows their descriptive statistics. Following (Elbaum et al., 2002), we used all the faulty versions whose faults can be revealed by more than 0% and less than 20% of the test cases.

For the **number of runs** per test case prioritization technique, according to (Arcuri and Briand, 2011), 1000 runs per subject are sufficient to evaluate a randomized technique on the subject. Following (Elbaum et al., 2002), we generated 1000 test suites iteratively from the test pool such that each test suite can cover all branches in the subject at least once. 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. Since both the LBS techniques and random were nondeterministic due to the impact of random seed, we set the **number of repeated trials** of each technique over each test suite to be 50.

Each of our subject programs accepts command line and file as inputs. We extracted the command line and file contents as strings, and use the edit distance (Gusfield, 1997) as **function** $\boldsymbol{f}_1$.

4.2.1. Independent variables

The first independent variable we studied was the **test case prioritization (TCP) technique**. We compared the family of LBS test case prioritization techniques with random and several classical search-based (total statement, additional statement, 2-Optimal, Hill Climbing, and Genetic Algorithm) techniques for test case prioritization as shown in Table 3. We selected them because these are representative search-based techniques for test case prioritization and they are evaluated in the experiment reported by Li et al. (2007). Our implementation of total statement and additional statement follows strictly according to the algorithms described in Elbaum et al. (2002). The same implementations for the greedy algorithms have been used to report the findings presented in Jiang et al. (2009).

We did not compare with those black-box input-based test case prioritization techniques such as Ledru et al. (2011) and Thomas et al. (2014) because of two reasons. The first reason was that the empirical studies in those work focused on the comparison of different test case distances while ignoring the comparison of test case prioritization effectiveness. In our controlled experiment, we aimed to compare our LBS techniques with classical search-based techniques directly. The second reason is that we aimed at systematically exploring the factor of test set distance, beam width, and candidate set size, which already made the scale of the experiment huge.

The second independent variable was **test set distance**. Specifically, we aimed to compare whether using a fixed distance measure (LBS_{100}, LBS_{010}, and LBS_{001}) shared the same effectiveness and efficiency as using a mixed distance measure (o LBS_{110}, LBS_{011}, and LBS_{101}). We also examined a follow-up question to study whether using a composite distance measure increases the effectiveness further (i.e., LBS_{111} versus others).

We had also evaluated all combinations of the **candidate set size** c and **beam width** k systematically where $c \in \{10, 20, 30, 40, 50\}$ and $k \in \{3, 4, 5, \ldots, 20\}$. We aim at examining whether the choice of specific value combinations of c and k may affect the same technique (i.e., holding the value of other independent variables the same) significantly. For these 90 combinations of k and c, we performed 225,000 prioritizations for each LBS technique on each subject, resulting in 6.3-million permutations of test suite in total to study *RQ1* and *RQ3*.

For *RQ2*, we randomly selected x%, where $x \in \{10, 20, 30, 40, 50, 60\ 70, 80, 90, 100\}$ of the whole test pool of each subject as a test suite. We applied each such test suite to each technique and measure the time cost (see the next section). We repeated this procedure 1000 times.

4.2.3. Dependent variables

We used ***APFD*** to measure the rate of fault detection (Elbaum et al., 2002). *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 equation:

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

Finally, to measure the efficiency of different test case prioritization techniques, we used the **time cost for test case prioritization** (in seconds).

4.3. Results and analysis

We find that different combinations of k and c produced more or less similar results, to avoid overloading readers by similar graphs, we report a representative case for *RQ1* and *RQ2* in this paper, which is $c = 10$ and $k = 7$. For *RQ3*, we summarize the results on all possible value combinations of k and c. From the result for *RQ3*, readers can observe that the combination of $c = 10$ and $k = 7$ is located well within the distributions to show in the last subsection.

4.3.1. Answering RQ1

For each technique, we calculated the APFD results across all the faulty versions and draw box-and-whisker plots on each UNIX program as shown in Figs. 1–4. The x-axis shows the different test case prioritization techniques and the y-axis shows APFD values.

From Figs. 1–4, we find that the LBS techniques perform well: they are more effective than both random and total greedy significantly on

Table 2
Benchmark suite.

Subject	Description	No. of faulty versions	Executable line of code	Real-life program versions	Test pool size
flex	Lexical Analyzer	21	8571–10124	2.47–2.54	567
grep	Text Searcher	17	8053–9089	2.4–2.4.2	809
gzip	File Compressor	55	4081–5159	1.1.2–1.3	217
sed	Stream Editor	17	4756–9289	1.18–3.02	370

Table 3
Prioritization techniques.

Name	Brief description
Random	Randomly order test cases
Total statement (total-st) (Elbaum et al., 2002)	Descending order of the total number of statement covered
Additional statement (addtl-st) (Elbaum et al., 2002)	Descending order of the coverage of statements not yet covered
2-Optimal (2_OPTI) (Lin, 1965, Skiena, 1998)	Select the next 2 test cases with largest additional coverage in each round
Hill Climbing (HILL) (Li et al., 2007)	Steep ascent: move to the state with largest increase in fitness
Genetic Algorithm (GA) (Holland, 1975)	Adaptive search based on natural genetic selection
LBS	Test Set Distance (f_2)
LBS_{100}	Eq. (1)
LBS_{010}	Eq. (2)
LBS_{001}	Eq. (3)
LBS_{110}	Eq. (4)
LBS_{101}	Eq. (5)
LBS_{011}	Eq. (6)
LBS_{111}	Eq. (7)

each program. 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% across all subjects. Since the theoretical optimal prioritization result reported in (Elbaum et al., 2002) is around 0.99, the LBS techniques are only 4% away from the optimal prioritization in terms of APFD.

Then we examine the results for the seven LBS techniques to evaluate the impact of using different test set distances. Among the LBS techniques, LBS_{110} performs best, followed by LBS_{100} and LBS_{010}. They consistently perform more effectively than other four LBS techniques. Interestingly, all the latter four LBS techniques involve maximizing the maximum distance in some or all beam steps. This indicates that maximizing the minimum or average distances may be a better selection criterion to spread the test case sequence more evenly. Maximizing the maximum distance is comparably less effective.

After analyzing the impact of test set distance, we further compare the best three LBS techniques (LBS_{110}, LBS_{100} and LBS_{010}) with those search-based techniques. We find that all these three LBS techniques perform significantly better than Hill-Climbing technique on each subject program in terms of median values. When compared with additional statement, 2-Optimal and Genetic Algorithm, the median values of LBS_{100}, LBS_{010}, and LBS_{110} are higher for all subject programs.

Following (Jiang et al., 2009), we further perform both the ANOVA test and the multiple comparisons between each pair of techniques

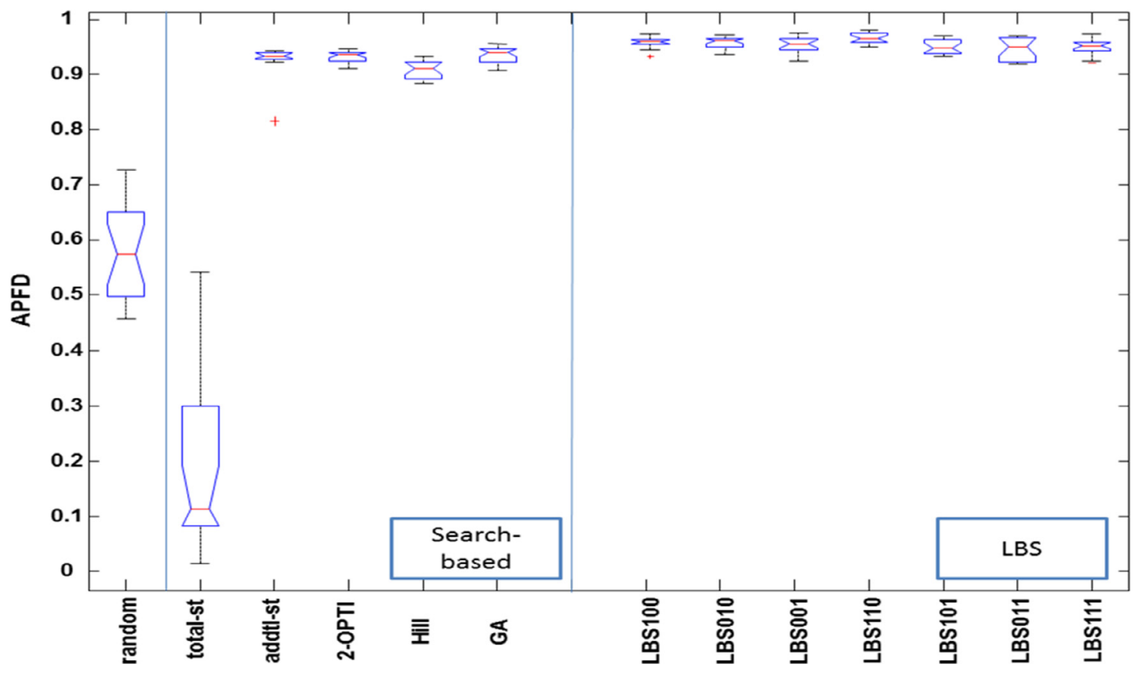

Fig. 1. APFD Results for *gzip*.

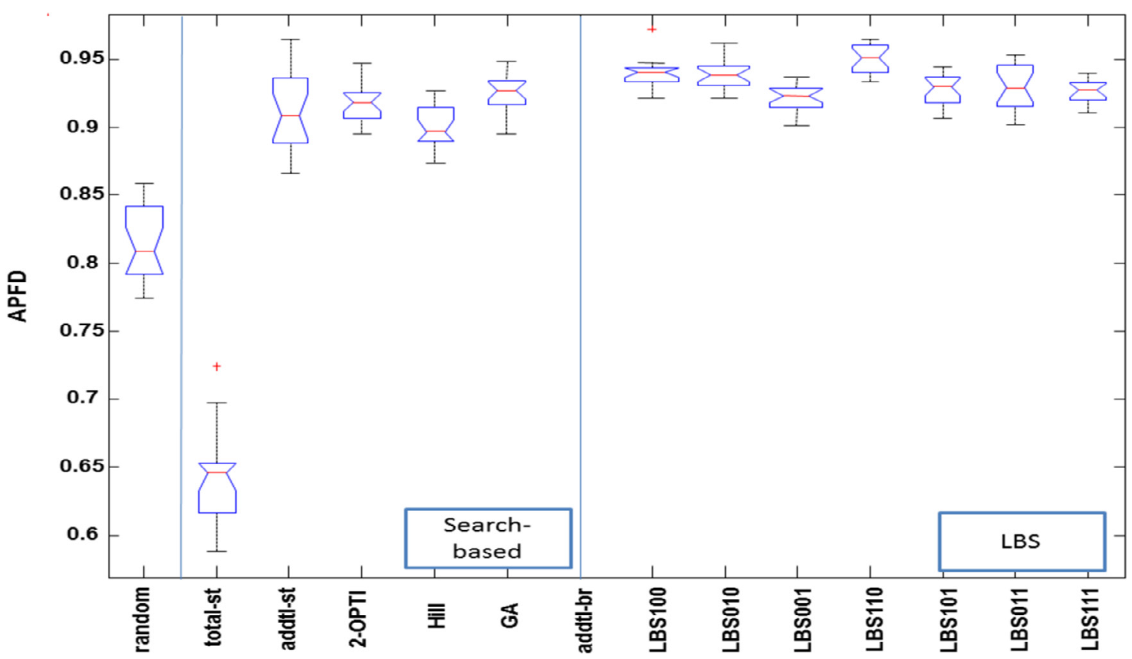

Fig. 2. APFD Results for *sed*.

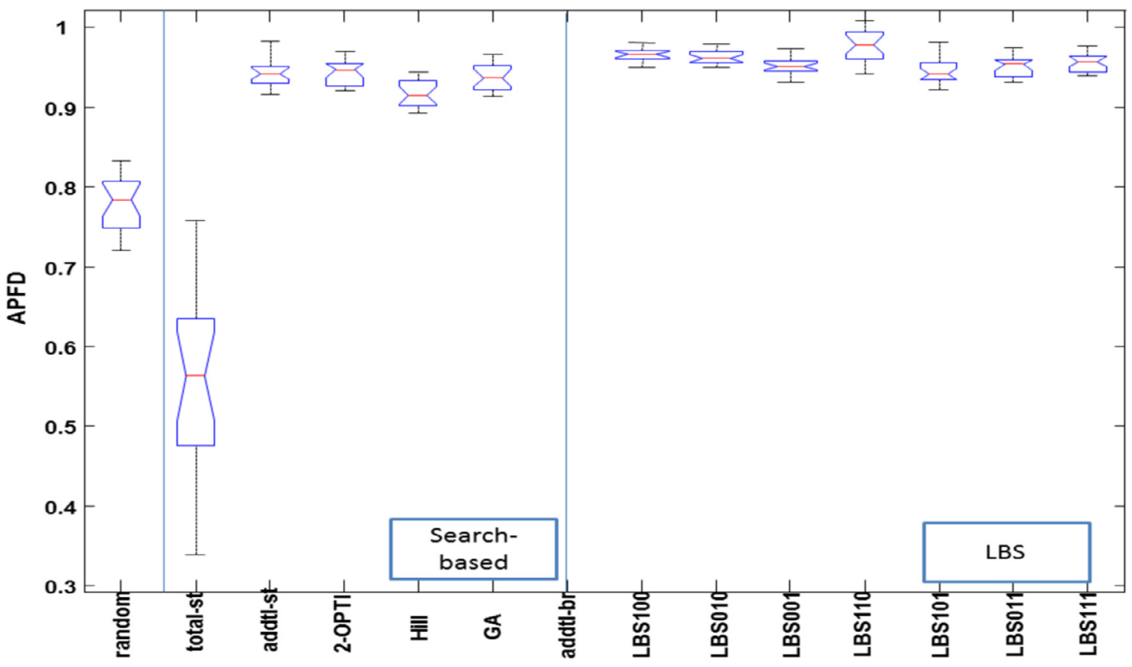

Fig. 3. APFD Results for *flex*.

to check whether their means are different significantly on each program. The results are shown in Figs. 5–8.

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

We also find that LBS_{100}, LBS_{010}, and LBS_{110} are more effective than the other search-based techniques, but the difference may not always be statistically significant.

As shown by the two dotted (blue) lines in Figs. 5–8, the LBS techniques as a whole are not different in a statistically meaningful way from the Additional Greedy technique, 2-Optimal and Genetic Algorithm at the 5% significance level. At individual technique level, LBS_{110} (i.e., the best LBS technique) is observed to achieve higher mean values (i.e., the central points in these bars) than all the other techniques on all four subjects, even though the differences are not statistically significant at the 5% significance level.

To answer RQ1: We observe that the LBS family of techniques can achieve higher mean effectiveness than, if not as effective as, the additional greedy, 2-Optimal, Hill Climbing, and Genetic Algorithm. Among all techniques studied in the controlled experiment, we observe that the most effective technique is $LBS_{110.}$ However, the pairwise differences among the LBS techniques and other techniques in the experiment are not large enough to make these differences to be statistically significant at the 5% significance level. We also observe that all techniques often achieve higher than 90% APFD on average.

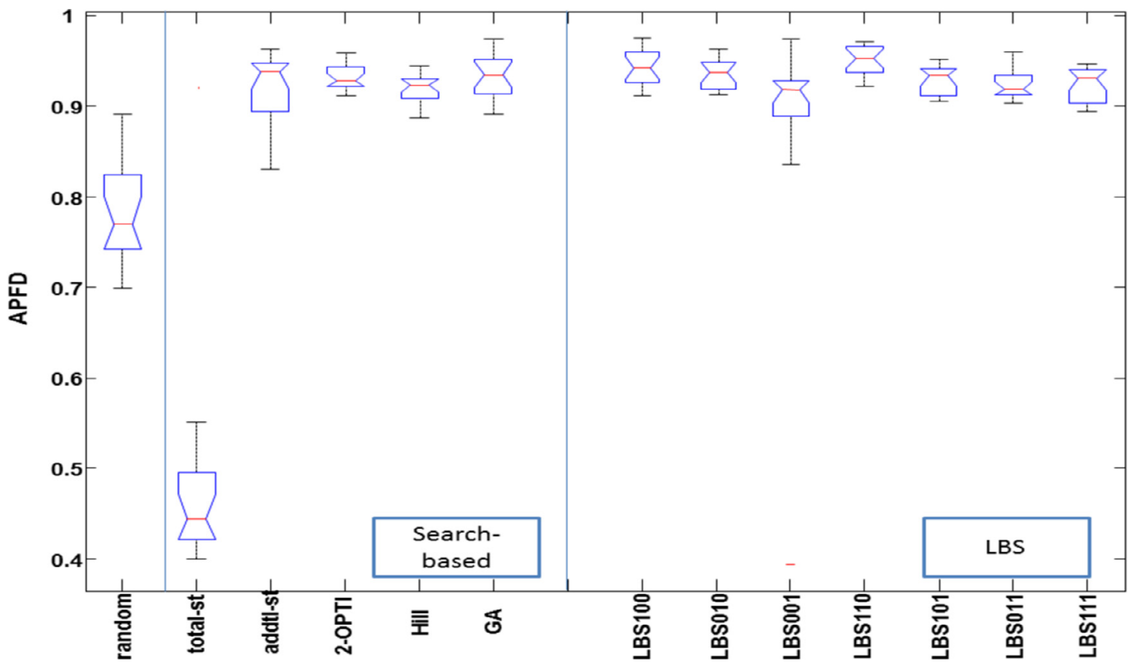

Fig. 4. APFD Results for *grep*.

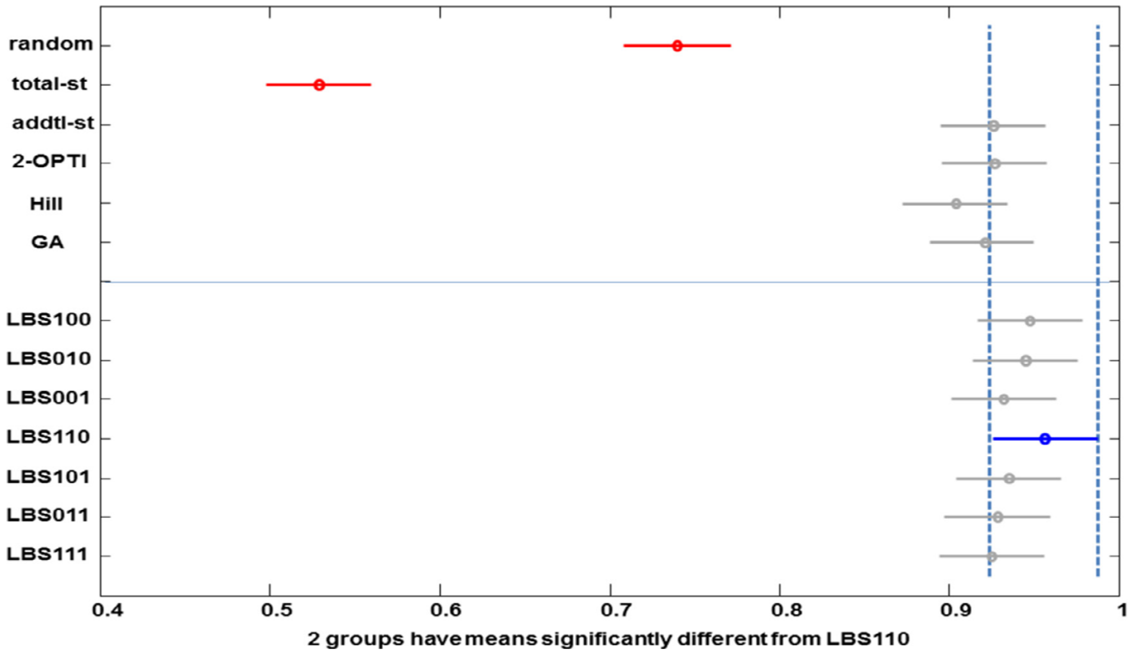

Fig. 5. Multiple comparison between LBS, search-based algorithms and random in terms of APFD on *gzip*. (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

4.3.2. Answering RQ2

In this section, we analyze the data for the input-based LBS techniques with random and classical search-based techniques using code coverage 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 each evaluated prioritization techniques, and record the prioritization time.

As shown in Fig. 9, 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 time costs of different techniques vary a lot, we show the *y*-axis up to 100 s so the curve for most techniques can be shown clearly.

In general, we observe that the LBS techniques only use slightly more time than random and are more efficient than those search-based techniques. We observe that the time cost of the additional statement technique grows fast when the size of test suite increases. For example, when using around 20% of the test pool, the prioritization time cost of addtl-st is slightly more than 25 s, but it reaches 700 s if using 100% test cases in the test pool (not shown in the figure). The next most costly techniques in turn are GA, 2-Optimal, Hill-Climbing, and Total-st. Following them, the prioritization time used by input-based LBS techniques overlaps with each other. Finally, random is most efficient. Furthermore, the time costs of the LBS techniques grow much slower than all other search-based techniques

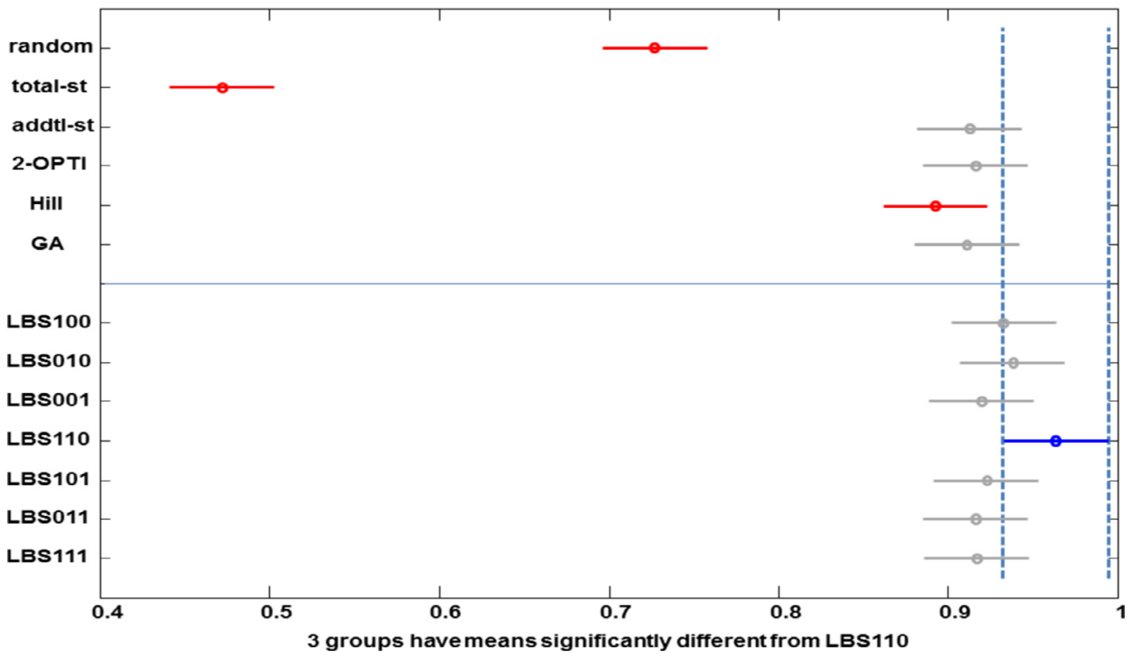

Fig. 6. Multiple comparison between LBS, search-based algorithms and random in terms of APFD on *sed*. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

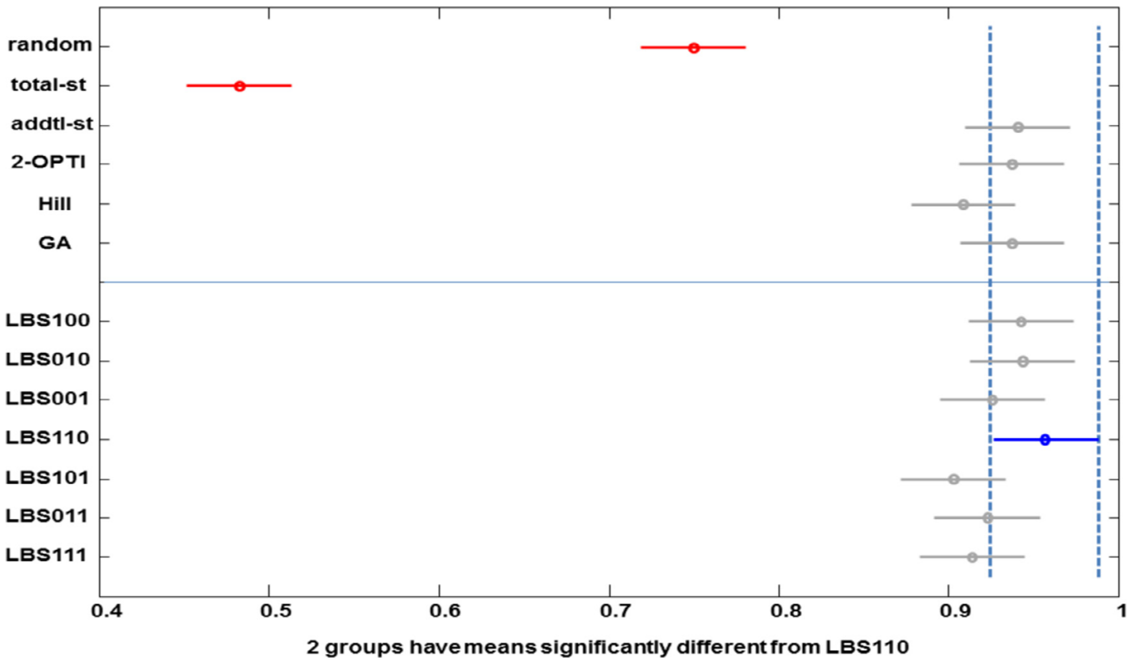

Fig. 7. Multiple comparison between LBS, search-based algorithms and random in terms of APFD on *flex*. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

as the size of a test suite increases. Note that we average the results over the four programs due to space limitation, but the trend on each program is the same.

We can answer RQ2 that the input-based LBS prioritization techniques can be efficient techniques for real-life medium-scale applications when compared with the other search-based techniques. Combined the answers to RQ1 and RQ2, we can conclude that the input-based LBS techniques can be as effective as the best search-based techniques using code coverage information and yet are much more efficient.

4.3.3. *Answering RQ3*

In this section, we study whether the size of candidate set and the beam width are important factors affecting the effectiveness of the LBS techniques. We use all seven LBS techniques to evaluate these two factors and report our finding. The results are shown in Figs. 10–13. The *x*-axis represents the 90 $(= 5 \times 18)$ combinations of *k* and *c*, and the y-axis represents *APFD* values. The seven lines in each figure correspond to each LBS technique.

We observe that the effectiveness in terms of *APFD* is not significantly affected by the size of the candidate set and beam width on

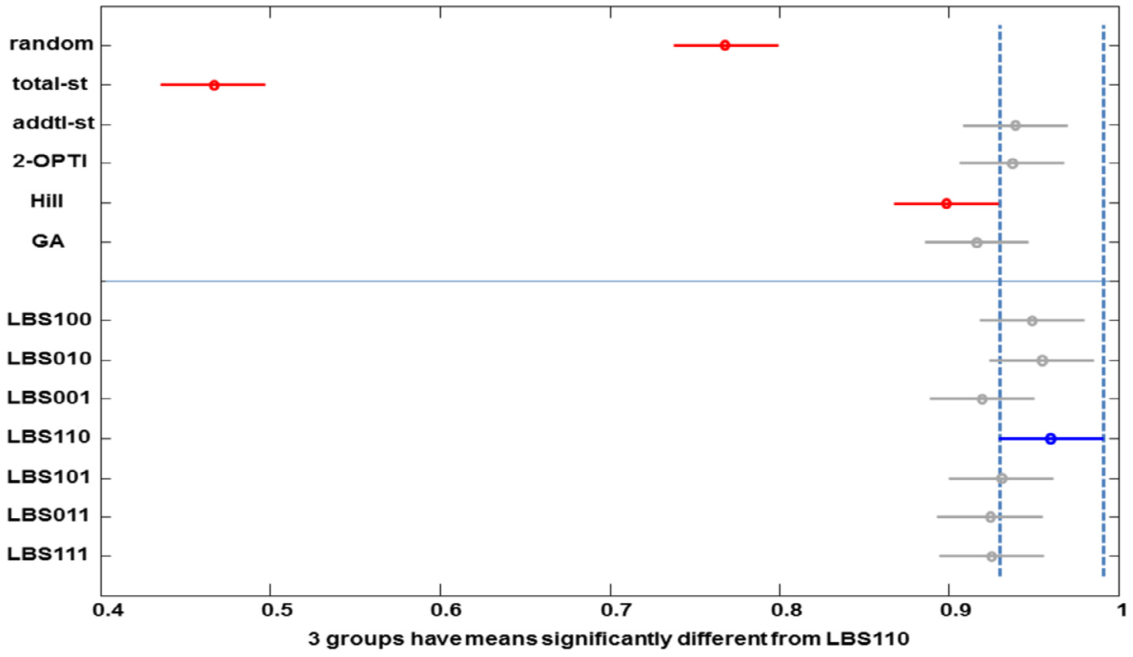

Fig. 8. Multiple comparison between LBS, search-based algorithms and random in terms of APFD on *grep*. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

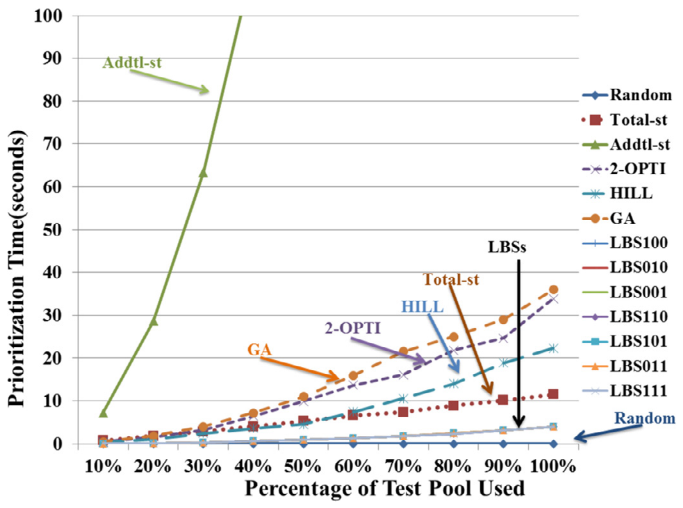

Fig. 9. Efficiency of different techniques.

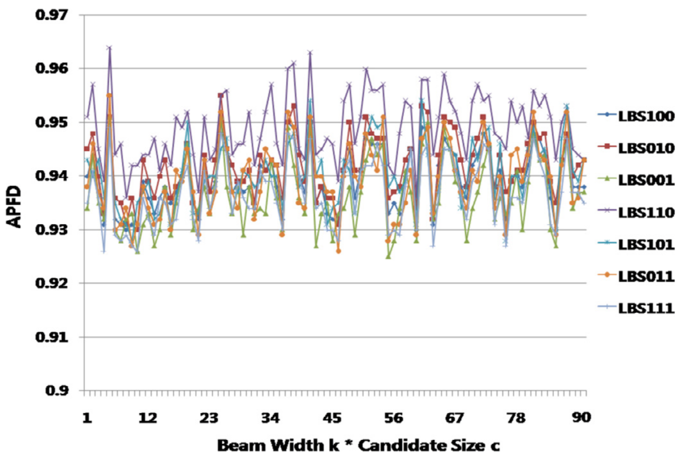

Fig. 11. Impact of candidate set size and beam width on APFD for *sed*.

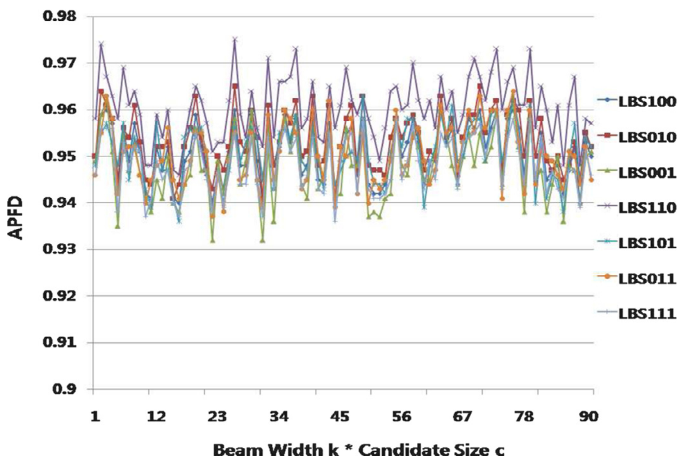

Fig. 10. Impact of candidate set size and beam width on APFD for *gzip*.

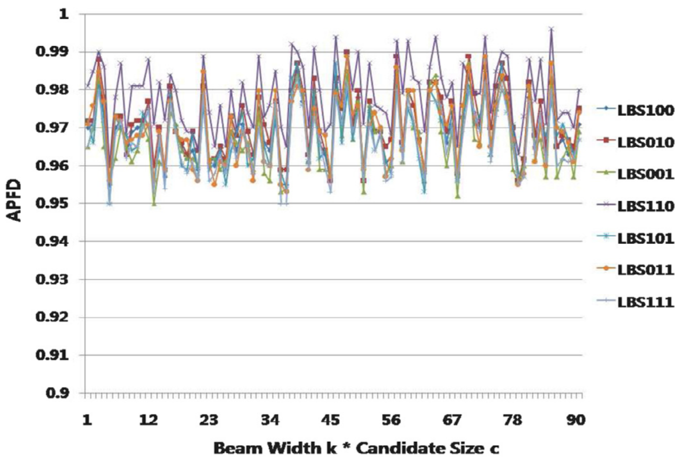

Fig. 12. Impact of candidate set size and beam width on APFD for *flex*.

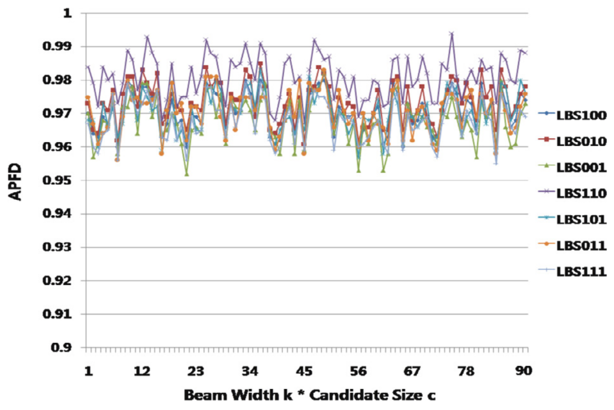

Fig. 13. Impact of candidate set size and beam width on APFD for *grep*.

each of the four subjects. In fact, for each of the subject programs and each combination of *c* and *k*, the *APFD* results only show relatively small variations. The *APFD* results for *gzip* ranges from 0.93 to 0.98; the results for *sed* ranges from 0.92 to 0.97; the results for *flex* ranges from 0.95 to 0.998; and the results for *grep* ranges from 0.95 to 0.995. We have also performed the ANOVA test on the data for each subject to check whether different combinations of *c* and *k* differ significantly from each other for the same LBS technique.

We consistently get small p-values of ranging from 0.000023 to 0.0066 for each LBS technique on each of the four subjects. This level of p-value consistently and successfully rejects the null hypothesis that different combinations of *c* and *k* differ significantly from each other at a significance level of 5% in affecting the effectiveness of the same LBS technique.

We can answer RQ3 that both candidate set size and the beam width size have no significant impact on the effectiveness of the LBS techniques. Thus, the test engineers may choose to use a relatively smaller candidate set size and a relatively small beam width to achieve better efficiency without compromising effectiveness, which is encouragingly.

4.4. Threats to validity

Our platform only supported the evaluation of the selected C programs. A controlled experiment on subject programs written in other programming languages can cross-validate our findings. We only used the branch-adequate test suites to conduct our controlled experiment. The way we generated branch-adequate test suite was the same as the procedure reported in (Do et al., 2008), and the use of branch-adequate test suites was popularly used in test case prioritization research studies (Elbaum et al., 2002; Jiang et al., 2009). In practice, there are other kinds of test suites. A further study on different kinds of test suites will 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. To reduce bugs in our framework, we have carefully performed code inspection and testing to assure correctness.

We downloaded the test pool from an open-access repository to construct test suites. The failure rates of different test pools may be different, which may affect the comparison results and the APFD values obtained by each technique.

There are many distance metrics for measuring the distances between program inputs. For example, the Hamming distance evaluated in (Ledru et al., 2011) was proposed by Hamming for binary tuples, but was generalized to cover tuples of the form $<$*field*$_1$, *field*$_2$, ..., *field*$_u$$>$ such that the number of possible values in each *field*$_i$ is finite. For the inputs of subjects under study, we believe string edit distance

Table 4
Prioritization techniques evaluated in the case study.

Name	Brief description
Random	Randomly order test cases
Greedy	Adapted Greedy algorithm in our setting
ART	Adapted Adaptive Random Testing algorithm in our setting
Genetic Algorithm (GA)	Adapted Genetic algorithm in our setting
LBS	Test Set Distance (f_2)
LBS$_{100}$	Eq. (1)
LBS$_{010}$	Eq. (2)
LBS$_{110}$	Eq. (4)

is more appropriate for the test case distance. We will investigate this aspect as a future work.

The LBS algorithm is also a randomized algorithm. To mitigate the impact of random seeds, we repeated the test case prioritization procedure with our LBS techniques (and random) 50 times on each test suite to report the mean of the obtained values. We note that similar practice is also reported in Thomas et al. (2014).

Similar to previous work (Elbaum et al., 2002; Ledru et al., 2011; Thomas et al., 2014), we used APFD to evaluate test case prioritization techniques. If readers are interest in the rate of code coverage, readers may examine how fast the code coverage has grown through the metric APSC (Li et al., 2007). We measured the efficiency of a test case prioritization technique by its time cost, which is also adopted in Jiang et al. (2009).

We are aware that many experiments that evaluate search-based test case prioritization techniques do *not* present the results in terms of fault of rate detection. This makes us difficult to validate whether our results could be consistent with those published results, which poses a threat in interpreting our results on these techniques. In Figs. 5–8, readers may observe that GA could not outperform addlt-st using code coverage. This finding is consistent to the claim by Zhang et al. (2013) that Additional Greedy is still the most effective coverage-based strategy in terms of APFD.

5. Case study

In our controlled experiment reported in Section 4, we have compared our LBS techniques directly with several existing techniques for test case prioritization. In this section, we study whether the adoption of the LBS search algorithm per se is the key to the effectiveness of our LBS techniques. To achieve this goal, we compare our LBS search algorithm with other search-based algorithms in the same settings. More specifically, we compare it with different search algorithms, including the Genetic algorithm, the ART algorithm, and the Greedy algorithms as shown in Table 4. We make them share same test case encoding using the test **input data** and set **even spread of test cases within input domain** as the same optimization goal. The assessment is based on both effectiveness (APFD) and efficiency (time cost spent on prioritization). In this way, we can dissect our techniques to discern the very cause of its improvements and the corresponding trade-offs.

5.1. Algorithms for comparison

We have adopted three search-based algorithms, i.e., Genetic (Li et al., 2007), ART (Jiang et al., 2009), and Greedy algorithms for evaluation. Similar to LBS, for all three techniques below, we use the same input data (rather than white-box coverage information) as the test case encoding.

For the **Genetic algorithm**, we use the same procedure and configuration as described in Section 2.3.4, except that we use discrepancy (see Section 3.2) as the fitness function.

For the **ART algorithm**, we use the same algorithm as described in (Jiang et al., 2009), except we use input distance rather than code coverage-based test case distance. For test set distance, we choose to maximize the minimum distance between a candidate test case and the set of already selected test cases, because this choice is best among all options (Jiang et al., 2009).

For the **Greedy algorithm**, we try to select the current best (highest discrepancy) subsequence of test cases in each round. Note that discrepancy is a single value rather than a vector, which is different from the traditional total and additional test case prioritization techniques. We begin with a subsequence with length one (i.e., only one test case), and we select one subsequence with largest discrepancy. Then in the next round, we build a new set of subsequences by adding each unselected test case into the current subsequence. We choose one best (having largest discrepancy) subsequence from the whole set. This process continues until all test cases are prioritized and the subsequence becomes the prioritized permutation of the whole test suite. Note that it is essentially a new algorithm proposed in this paper instead of the classical greedy algorithm or additional greedy algorithm.

For the **LBS** algorithm, we choose LBS_{100}, LBS_{010}, and LBS_{110} for evaluation because they perform consistently well within the LBS family presented in Section 4.

We also compare to **random** ordering in the case study.

5.2. Setup of the case study

In this subsection, we outline the setup of our case study. We used the same hardware platform as in our controlled experiment. We also used the same set of UNIX subject programs from SIR. We randomly selected another 50 suites from the 1000 branch-adequate test suites for evaluation. Since the Genetic, ART, and LBS algorithms all involved randomness, we also set the number of repeated trials for each technique as 50 as what we did in (Jiang et al., 2009).

To compare different algorithms in the same setting, we used the same test case encoding. As in our controlled experiment, we extracted the command line and file contents as strings to encode a test case. Furthermore, we also used the edit distance (Gusfield, 1997) for measuring test case distance. We used APFD to measure the effectiveness of different test case prioritization techniques and use prioritization time to measure their efficiency.

5.3. Results and analysis

We present the results of our case study in this section.

First, we show the results in terms APFD for different test case prioritization techniques over all four UNIX programs. As shown in the box-whisker plot in Fig. 14, the x-axis represents the search-based algorithms used in the corresponding test case prioritization techniques; the y-axis shows their APFD values. We observe that the notches of both random and Greedy techniques do not overlap with ART, GA, and LBS techniques, which indicates that their median values differ from each other at the 5% significance level. The notches of GA and LBS techniques also do not overlap. On the other hands, the ART technique overlaps with LBS_{100} and LBS_{010} techniques, but not with LBS_{110}, which means LBS_{110} performs best among all techniques, followed by LBS_{100}, LBS_{010}, and ART.

We further compare the mean values of these techniques using the multiple comparison procedure as shown in Fig. 15. We observe that both random and Greedy techniques have means significantly different from LBS_{110}, which indicates that the LBS techniques perform significantly more effective than them at the 5% significance level. On the other hand, the means of LBS, GA, and ART techniques do not differ from each significantly at 5% significance level, although LBS techniques do performs a little better than ART and GA in terms of mean values.

Having compared the effectiveness of different search-based techniques, we further compare their efficiency as shown in Fig. 16. The x-axis is the percentage of test pool used and the y-axis is the time

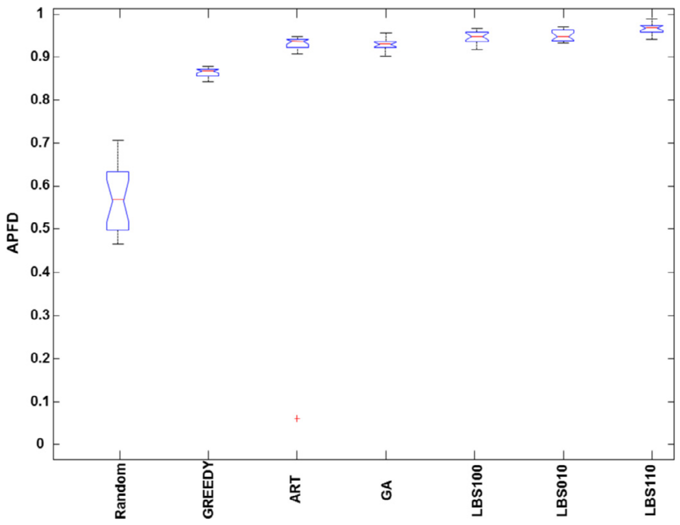

Fig. 14. Distribution of different search-based algorithms with the same setting.

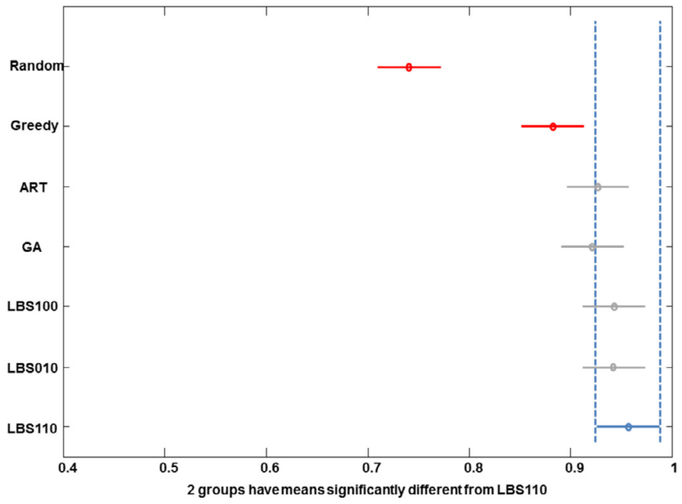

Fig. 15. Multiple comparisons of different search-based algorithms with the same setting.

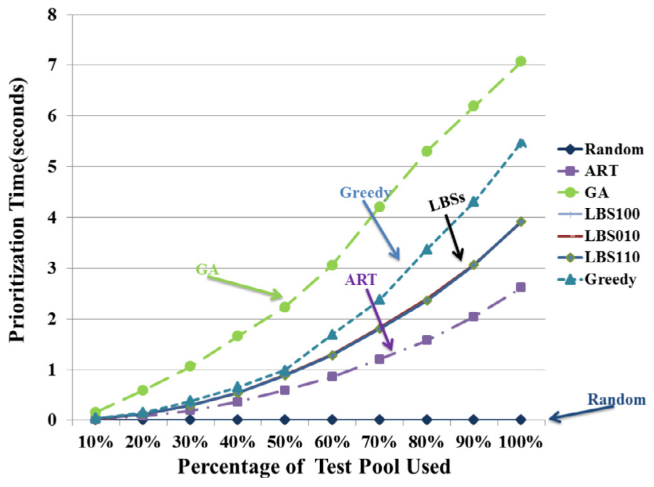

Fig. 16. Efficiency comparison of different search-based algorithms with the same setting.

used for test case prioritization averaged over all four UNIX programs. The y-axis shows the time spent on test case prioritization.

In general, the prioritization times of different techniques in our settings are small. This is especially clear when comparing the prioritization time for GA and Greedy between Fig. 16 and Fig. 9: the adoption of input data as test case encoding and the even-spread of test cases as optimization goal reduce the prioritization cost significantly.

We also find that ART uses less time than LBS techniques, which may be due to its limited search width. The LBS techniques still overlap each other. They are slightly more efficient than Greedy and GA techniques. Although the Greedy technique only keeps one subsequence in each iteration, it evaluates all possible combinations in each iteration as well. In contrast, although the LBS techniques keep multiple subsequences for next iteration, it only evaluates those subsequences from the candidate set. It seems a small beam width and candidate set size can lower the time cost. Finally, the Genetic technique incurs the highest cost among all techniques, which may be due to its slow convergence.

To summarize, when we compare different algorithms with the same input data and heuristic, the LBS, ART, and Genetic algorithms are significantly more effective than the Greedy algorithm. Moreover, the three LBS techniques achieve higher mean or medium APFD values than ART, and Genetic, and yet the differences are not statistically significant at the 5% significance level. Among the prioritization costs of all studied techniques in the current setting, LBS incur a higher cost than ART, and it incurs a lower cost than both Greedy and Genetic.

After comparing different search-based algorithms in the same setting, we can conclude that the improvement of LBS techniques over existing techniques is mainly due to the use of our randomized Local Beam Search algorithm rather than the use of input data and the optimization goal. Meanwhile, as shown in the results on the comparison of efficiency, the adoption of input data and the strategy of evenly spread of test cases over the input domain do help reduce the prioritization cost significantly when compared with other combination of test case encoding and optimization goal.

6. Further discussion

6.1. Application domains

In this section, we discuss the application domain of our LBS techniques. In general, like existing ART techniques, the LBS techniques are applicable to any application as long as the input-based test case distance can be defined. Admittedly, the inputs-based test case distance for different applications varies significantly due to the variety of the structures of test inputs. In this work, our test case distance is defined against the inputs of UNIX core utility programs with numerical and string format (i.e., Euclidean and string edit distance). Thus, our LBS techniques are directly applicable to applications with numerical and string inputs. Our LBS approach can also be extended to test other applications by defining the test case distance based on their corresponding input formats. For example, to test event-driven application using an LBS technique, it is necessary to define the distance between event sequences. If testers have no knowledge about the selection or definition of test case distance, a further controlled experiment on some generic distances may ease testers. We leave it as a future work.

Moreover, from the empirical results in answering *RQ2*, the prioritization time cost for LBS techniques was consistently less than 10 s. This shows that LBS techniques can be practical for use in terms of efficiency.

6.2. Test cases

Our work also suffers from the limitations in handling very large input datasets. The cost of our LBS techniques can be reduced further, however. For instance, if a test suite applicable to a program under test is large, one may further set the parameters m and k in our techniques 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 an NP-hard search problem is to apply approximation algorithms, which is an interesting future work.

Apart from the test input size dimension, the execution profiles of the program over these inputs may also have an impact on the relative merits between input-based and code-coverage-based techniques. A further experimentation is needed to clarify the situation. Intuitively, there are situations where either approach is better. A clarification can help to provide a hybrid or adaptive approach to achieve a more cost-effective result.

Our algorithm computes the even spread of test cases through a discrepancy metric. Without the computation of discrepancy, the algorithm cannot ensure the selected successors can be both faraway from each other and distributed within the input domain with equal density. Alternatively, by substituting this metric by some other metrics, the effectiveness of LBS techniques will also change. It is interesting to understand the impact of such a metric by adapting our techniques further in future work.

6.3. Evolutionary algorithms

There is a recent trend in software engineering research community to apply evolutionary algorithms (e.g., GA in our experiment) to address software engineering problems. In our experiment, GA works consistently inferior to both existing techniques and our proposed algorithms: Greedy, ART, and LBS if the goal is the rate of fault detection. The time cost for GA is also less attractive than ART or LBS. We recall that in evolutionary computing, there is a famous conjecture, informally known as the *No Free Lunch theorem* (Wolpert and Macready, 1997). Its essence is that if a kind of evolutionary algorithm is effective in one application domain, then there is another domain that it is ineffective. It is unclear whether test case prioritization with the goal of fault detection is an ineffective application domain to apply GAs.

Some readers may conjecture that on some other benchmarks, GA may be superior. We are unware of strong evidences in the literature yet. It is interesting for the search-based software engineering community to find out the case.

From what we observed from the result of our experiment, GA appears to over-fit toward a given heuristic. Unfortunately, such a heuristic is unable, even in theory, to be a perfect encoding of the ultimate goal of software testing fault detection in general, and the rate of fault detection in our study. Is evolutionary algorithm for optimization an answer of a software engineering problem that could only be approximated? We leave the answer open to readers.

7. Related work

In this section, we review the closely related work.

Input-based or output-based test case prioritization is not completely new. Mei et al. (2015a) 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. (2014) 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. Mei et al. (2012) proposed a static approach to prioritizing JUnit test case, which analyzes the static call graphs of JUnit test cases and the program under test to estimate the ability of each test case to achieve code coverage, and then prioritize the test cases based on the estimation. Our technique, however, is not a specialized technique for JUnit test cases. Hao et al. (2013) proposed a test case prioritization technique that dynamically prioritizes the test cases by consulting the execution output of the already prioritized test cases. In contrast, our technique requires no dynamic execution information and thus the prioritization process has no impact on execution efficiency.

Thomas et al. (2014) proposed a new static black-box test case prioritization technique that represents test cases using the linguistic data of the test cases. They applied a text analysis algorithm called topic modeling to the linguistic data to approximate the functionality of each test case, allowing their technique to give high priority to test cases that test different functionalities of the SUT. In (2011), Ledru et al. also proposed a static black-box test case prioritization algorithm to maximize test case diversity. They compared four different test case distances: Euclidean, Manhattan, Levenshtein, and Hamming distance. Similar to these two pieces of work, our work also tries to maximize test case diversity through black-box input information. Our work differ from them in two aspects: we use a randomized beam search algorithm to explore the search space and we target at systematically comparing the impact of different test set distance rather than test case distance.

Jiang et al. (2009) proposed code coverage based adaptive random test case prioritization. Compared to the result reported in Jiang et al. (2009), the result of our input-based LBS techniques is not different in a statistically meaningful way from their code coverage based techniques. Zhou (2010) proposed to use coverage information to guide test case selection in ART, his work is similar to Jiang et al. (2009) except that it uses the Manhattan distance to measure the distance between test cases.

In (2012), 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. (1997) combined test suite minimization and prioritization techniques to select cases based on the cost per additional coverage. Walcott et al. (2006) used GAs to reorder test cases under time constraints such that their techniques can be time-aware. Zhang et al. (2009) used the integer linear programming technique to find an optimal coverage solution, which can lead to effective test case prioritization results. Srivastava and Thiagarajan (2002) 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. Kim and Porter (2002) proposed a history-based test case prioritization technique for regression testing in resource constrained environments. They use the ideas taken from statistical quality control and statistical forecasting to improve test case prioritization effectiveness. In (2011), Carlson also proposed new prioritization techniques that incorporate a clustering approach and utilize code coverage, code complexity, and history data on real faults. Their results show that clustering can help improve the effectiveness of test case prioritization.

Li et al. (2007) 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. The experiment presented in this paper further shows that these search-based algorithms also do not perform better than additional greedy algorithms in terms of APFD. You et al. (2011) evaluated time-aware test case prioritization on the Siemens suite and the program *space*. They found that the differences among techniques in terms of AFPD were not statistically significant. Qu et al. (2007) proposed to group test cases according to their failure-exposing history and to adjust their priority dynamically during executions. Although their techniques are black box ones, they require execution history information that may not be available.

We also conjecture that the limitations of ART mentioning in Arcuri and Briand (2011) are no longer applicable in Arcuri et al. (2012) because unlike the former work, the latter work is no longer referencing ART (Chen et al., 2004). For brevity, the present paper does not discuss these once-mentioned limitations in the evaluation of the experiment presented in Jiang et al. (2009).

Most of the above-reviewed test cases prioritization techniques were proposed with intuition and their relationships are often compared empirically. On the other hand, Mei et al. (2015b) proposed a novel refinement-oriented level-exploration (ROLE) strategy and presented the first provable hierarchy of test case prioritization techniques for testing of services. The ROLE strategy systematically includes additional dimensions of the coverage dataset when a base prioritization strategy cannot distinguish test cases at the current dimensions of the same coverage dataset. Based on the ROLE strategy, a technique at a higher level will only generate a chosen subset of prioritized test suites generated by another technique at a lower level, resulting in a refinement relation between the two techniques in their hierarchy. It is interesting to deepen our knowledge on the theoretical aspect of regression testing in the future.

8. Concluding remarks

Many existing test case prioritization techniques permute a test suite with the intention to improve the rate of fault detection via some heuristics. Furthermore, many 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. Static analysis on the source code of the program under test to get the approximated code coverage of individual test cases on the program under test can be imprecise. 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).

In this paper, we have developed a novel family of input-based test case prioritization techniques. Our LBS techniques effectively prune the 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 width of the local beam search.

We have reported a controlled experiment in this paper. The evaluation has shown that the input-based LBS techniques can be as effective as the best search-based techniques using code coverage information (including additional greedy, 2-Optimal, Hill Climbing and Genetic Algorithm) and are much more efficient than them. The best technique is $\text{LBS}_{110,}$ which achieves highest mean AFPD values than all the other techniques studied in the controlled experiment. In terms of test set distance, our controlled experiment has shown that randomly maximizing the minimum or average distance in each beam step is a better choice. It also found that the choice of beam width and candidate set size makes no significant impact on prioritization effectiveness. This indicates a smaller beam width and candidate set size can be a better choice for efficiency.

We have further reported a case study to compare the LBS algorithm with existing search algorithms adapted in the same setting (i.e., the same test case encoding and optimization goal). We have selected the three techniques that have been shown to be more effective in the above controlled experiment and used them in this case study. The data analysis has shown that the three LBS techniques achieve higher mean APFD values than both ART and GA. Moreover, all three LBS techniques are significantly more efficient than GA and yet are significantly less efficient than ART. The results have showed that the improvement of our LBS techniques over the other studied techniques is mainly due to the use of randomized Local Beam Search algorithm per se. Furthermore, the adoption of input data and the optimization heuristic (evenly spread the test cases over the input domain) do help reduce the prioritization cost significantly when compared with other combination of test case encoding and optimization goal.

In future work, we will investigate further generalization of the LBS techniques and ART techniques for test case prioritization. Our techniques currently use the distance measures such as string edit distances. In mathematics, these distances are *universal* and can measure the distance between any two sequences of tuples. It is however unclear to us whether the use of other distances may produce other surprising and good results. It is certainly provoking if formal analysis can be conducted. Our work adopts the idea of even spread of test cases in the spirit of adaptive random testing for test case generation. It is unclear to what extent the use of other notions or measures to assess the distribution over sequences of test cases can be effective and efficient.

References

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), pp. 1–10.
Arcuri, A., Briand, L., 2011. Adaptive random testing: An illusion of effectiveness? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11), pp. 265–275.
Arcuri, A., Iqbal, M.Z., Briand, L., 2012. Random testing: theoretical results and practical implications. IEEE Trans. Softw. Eng. 38 (2), 258–277.
Carlson, R., Hyunsook, D., Denton, A., 2011. A clustering approach to improving test case prioritization: an industrial case study. In: Proceedings of the 27th IEEE International Conference on Software Maintenance, pp. 382–391.
Chen, T.Y., Leung, H., Mak, I.K., 2004. Adaptive random testing. In: Advances in Computer Science: Proceedings of the 9th Asian Computing Science Conference (ASIAN 2004), 3321. Springer Lecture Notes in Computer Science, pp. 320–329.
Chen, T.Y., Kuo, F.-C., Liu, H., 2007. Distribution metric driven adaptive random testing. In: Proceedings of the 7th International Conference on Quality Software (QSIC 2007), pp. 274–279.
Do, H., Mirarab, S., Tahvildari, L., Rothermel, G., 2008. 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, pp. 71–82.
Eclipse. 2013. http://www.eclipse.org/ (Last accessed: 2013.01).
Elbaum, S.G., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. 28 (2), 159–182.
Elbaum, S.G., Rothermel, G., Kanduri, S., Malishevsky, A.G., 2004. Selecting a cost-effective test case prioritization technique. Softw. Qual. Control 12 (3), 185–210.
FireFox. 2013. http://www.mozilla.org/en-US/firefox/new/ (Last accessed: 2013.01).
Flex. 2013. http://flex.sourceforge.net/ (Last accessed: 2014.01).
Foursquared. 2012. http://code.google.com/p/foursquared/ (Last accessed: 2012.11).
Gusfield, D., 1997. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press ISBN 0-521-58519-8.
Google Code. 2013. http://code.google.com/ (Last accessed: 2013.01).
Hao, D., Zhao, X., Zhang, L., 2013. Adaptive test-case prioritization guided by output inspection. In: Proceedings of the 37th Annual International Computer Software and Applications Conference (COMPSAC 2013), pp. 169–179.
Holland, J.H., 1975. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artifical Intelligence. University of Michigan.
Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H., 2009. Adaptive random test case prioritization. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), pp. 233–244.
Jiang, B., Tse, T.H., Grieskamp, W., Kicillof, N., Cao, Y., Li, X., Chan, W.K., 2011. Assuring the model evolution of protocol software specifications by regression testing process improvement. Softw. Pract. Exp. 41 (10), 1073–1103.
Jiang, B., Chan, W.K., 2013. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In: Proceedings of the 37th Annual International Computer Software and Applications Conference (COMPSAC 2013). IEEE Computer Society, Los Alamitos, CA, pp. 190–199.
Kim, J.M., Porter, A., 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th International Conference on Software Engineering (ICSE '02), pp. 119–129.
Lahiri, S.K., Vaswani, K., Hoare, C.A.R., 2010. Differential static analysis: opportunities, applications, and challenges. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10), pp. 201–2014.
Ledru, Y., Petrenko, A., Boroday, S., Mandran, N., 2011. Prioritizing test cases with string distances . Autom. Softw. Eng. 19 (1), 65–95.
Li, B., Sun, X., Leung, H., Zhang, S., 2013. A survey of code-based change impact analysis techniques. Softw. Test. Verific. Reliabil. 23 (8), 613–646.
Li, Z., Harman, M., Hierons, R.M., 2007. Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. 33 (4), 225–237.
Lin, S., 1965. Computer solutions of the travelling salesman problem. Bell Syst. Tech. J. 44, 2245–2269.
Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K., 2005. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005). ACM, New York, NY, USA, pp. 190–200.
Mei, H., Hao, D., Zhang, L.M., Zhang, L., Zhou, J., Rothermel, G., 2012. A static approach to prioritizing Junit test cases. IEEE Trans. Softw. Eng. 38 (6), 1258–1275.
Mei, L., Chan, W.K., Tse, T.H., Jiang, B., and Zhai, K. Preemptive regression testing of workflow-based web services. IEEE Trans. Serv. Comput. 2015. doi:10.1109/TSC.2014.2322621.
Mei, L., Cai, Y., Jia, C., Jiang, B., Chan, W.K., Zhang, Z., Tse, T.H., A subsumption hierarchy of test case prioritization for composite services. IEEE Trans. Serv. Comput. 2015. doi:10.1109/TSC.2014.2331683.
MySQL. 2013. http://www.mysql.com/ (Last accessed: 2013.01).
Onoma, A.K., Tsai, W.T., Poonawala, M., Suganuma, H., 1998. Regression testing in an industrial environment. Commun. ACM 41 (5), 81–86.
Qu, B., Changhai, N., Xu, B., Zhang, X., 2007. Test case prioritization for black box testing. In: Proceedings of 31st Annual IEEE International Computer Software and Applications Conference (COMPSAC 2007), vol. 1, pp. 465–474.
Rothermel, G., Harrold, M.J., 1997. A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. 6 (2), 173–210.
Rovegard, P., Angelis, L., Wohlin, C., 2008. An empirical study on views of importance of change impact analysis issues. IEEE Trans. Softw. Eng. 34 (4), 516–530.
Skiena, S.S., 1998. The Algorithm Design Manual. Springer- Verlag.
Srivastava, A., Thiagarajan, J., 2002. Effectively prioritizing tests in development environment. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), pp. 97–106.
Tassey, G., 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. In: National Institute of Standards and Technology report, RTI Project 7007.
Thomas, S.W., Hemmati, H., Hassan, A.E., Blostein, D., 2014. Static test case prioritization using topic models. Empirical Softw. Eng. 19 (1), 182–212.
Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S., 2006. Time aware test suite prioritization. In: Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), pp. 1–12.
Wolpert, D.H., Macready, W.G., 1997. No free lunch theorems for optimization. Trans. Evol. Comp. 1, 67–82.
Wong, W.E., Horgan, J.R., London, S., Agrawal, H., 1997. A study of effective regression testing in practice. In: Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE 1997), pp. 264–274.
Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verific. Reliabil. 22 (2), 67–120. doi:10.1002/stv.430.
You, D., Chen, Z., Xu, B., Luo, B., Zhang, C., 2011. An empirical study on the effectiveness of time-aware test case prioritization techniques. In: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC 2011), pp. 1451–1456.
Zhai, K., Jiang, B., Chan, W.K., 2014. Prioritizing test cases for regression testing of location-based services: metrics, techniques, and case study. IEEE Trans. Serv. Comput. 7 (1), 54–67. doi:10.1109/TSC.2012.40.
Zhang, L., Hou, S., Guo, C., Xie, T., Mei, H., 2009. Time-aware test-case prioritization using integer linear programming. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA 2009), pp. 213–224.
Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H., 2013. Bridging the gap between the total and additional test-case prioritization Strategies. In: Proceedings of 35th International Conference on Software Engineering (ICSE 2013), pp. 192–201.
Zhou, Z., 2010. Using coverage information to guide test case selection in adaptive random testing. In: Proceedings of IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW), pp. 208–213.

Bo Jiang is an assistant professor at the School of Computer Science and Engineering, Beihang University. He got his Ph.D. from the Department of Computer Science of The University of Hong Kong. His current research interests are software engineering in general and embedded software testing as well as program debugging in particular. He received the best papers award of COMPSAC'08, COMPSAC'09, and QSIC'11.

W.K. Chan is an assistant professor at the Department of Computer Science, City University of Hong Kong. He is on the editorial board of a few journals, including *Journal of Systems and Software*. He was the program chairs or track chairs of SETA 2015, APSEC 2012, QSIC 2010 and AST 2010, program/review committees of FSE'14 and ICSE'15, and guest editors of some journal special issues. His current research interest is in addressing the testing and analysis challenges in large-scale software systems.