Science of Computer Programming 214 (2022) 102730

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Candidate test set reduction for adaptive random testing: An overheads reduction technique

Rubing Huang [a], Haibo Chen [b,*], Weifeng Sun [c], Dave Towey [d]

[a] Faculty of Information Technology, Macau University of Science and Technology, Macau 999078, China
[b] School of Computer Science and Communication Engineering, Jiangsu University, Jiangsu 212013, China
[c] School of Big Data and Software Engineering, Chongqing University, Chongqing 401331, China
[d] School of Computer Science, University of Nottingham Ningbo China, Zhejiang 315100, China

A R T I C L E I N F O

Article history:
Received 29 January 2021
Received in revised form 27 September 2021
Accepted 27 September 2021
Available online 1 October 2021

Keywords:
Software testing
Random testing
Adaptive random testing
FSCS
Test set reduction

A B S T R A C T

Adaptive Random Testing (ART) is a family of testing techniques that were proposed as an enhancement of random testing (RT). ART achieves better failure-detection capability than RT by more evenly distributing test cases throughout the input domain. However, this process of selecting more diverse test cases incurs a heavy computational cost. In this paper, we propose a new ART method that improves on the efficiency of *Fixed-Size-Candidate-Set ART* (FSCS) by applying a test set reduction strategy. The proposed method, *FSCS by Candidate Test Set Reduction* (FSCS-CTSR), reduces the number of randomly generated candidate test cases, but supplements them with earlier, unused candidates that have lower similarity to the executed test cases. Simulations and experimental studies were conducted to examine the effectiveness and efficiency of the method, with the experimental results showing a comparable failure-detection effectiveness to FSCS, but with lower computational costs.

1. Introduction

Software testing is a fundamental activity in the software development lifecycle [1,2], playing an important role in software quality assurance. Among the black-box software testing methods, *Random Testing* (RT) [3] is a basic technique that randomly generates test cases within the input domain. Partly due to its simple concept and easy implementation, RT has been applied to a number of different applications [4]. RT has also been adopted as a baseline testing technique in empirical studies, such as regression testing [5], combinatorial testing [6], reliability testing [7], and integration testing [8]. However, some criticism about the effectiveness of RT relates to it not taking advantage of information of the system under test (SUT), such as the program code, or likely failure regions [9,10].

Previous research [11,12] has identified that, for a faulty program, failure-causing inputs tend to cluster into contiguous regions in the input domain. It has, therefore, been suggested that non-failure regions may also be contiguous: If a test case t fails to reveal a failure, then test cases that are similar to t may also not reveal any failures. Based on this observation, Chen et al. [13] proposed *Adaptive Random Testing* (ART). ART aims to increase the *diversity* [14] of test cases by more evenly

* Corresponding author.
E-mail addresses: rbhuang@must.edu.mo (R. Huang), 2221808034@stmail.ujs.edu.cn (H. Chen), weifeng.sun@cqu.edu.cn (W. Sun), dave.towey@nottingham.edu.cn (D. Towey).

https://doi.org/10.1016/j.scico.2021.102730
0167-6423/© 2021 Elsevier B.V. All rights reserved.

distributing them across the input domain. This increased diversity enables ART to achieve better failure-detection capability than RT [15,16].

A number of ART methods [17–24] have been proposed based on the notion of "even spreading", including the popular *Fixed-Sized-Candidate-Set Adaptive Random Testing* (FSCS) [25], which achieves very good failure-detection effectiveness [26]. However, FSCS also incurs heavy computational overheads, leading to criticism of it having low efficiency and limited practical application [27–29].

Previous analyses [17,25] of the FSCS algorithm have shown that the heavy computational overhead is mainly related to the similarity determination between candidates and executed test cases: To obtain a suitable test case, a fixed number of candidates are randomly generated, with the best one then selected. Determination of "best" is based on which candidate is considered to be the least "similar" to all previously-executed tests. Typically, the Euclidean distance is used as the similarity metric [30], requiring the distances from each candidate to each executed test case to be calculated. These distance calculations result in heavy computational overheads.

The distance calculation costs are significantly affected by the number of candidates. To alleviate the high computational overheads of FSCS, in this paper, we present a new strategy that reduces the number of candidate test cases involved in the calculations. While maintaining the candidate set size from the original FSCS (ten candidates), the strategy reduces the number of candidates that are *generated* in each round, thus lowering the number of new similarity comparisons needed. In each round, the candidate set comprises both newly-generated test cases, and some previously-generated, retained candidates. Using this strategy, at the end of each round, some similar candidates can be discarded, with the remaining ones being retained and passed to the next selection round (reducing the number of randomly-generated candidates). We call this test-set-reduction method *FSCS by Candidate Test Set Reduction* (FSCS-CTSR).

The results of our simulations and experiments with 19 real-world programs show that FSCS-CTSR can be an effective way to significantly reduce the FSCS computational overheads while maintaining the same level of failure-detection capability.

The main contributions of this paper are:

- We propose a cost-effective ART approach, FSCS-CTSR, to improve the efficiency of FSCS by reducing the number of candidate test cases.
- We examine the effectiveness and efficiency of FSCS-CTSR through simulations and experimental studies, reporting the results and statistical analysis in detail.
- Compared with FSCS, we show that FSCS-CTSR achieves similar effectiveness to FSCS, but significantly reduces the computational costs.

The rest of this paper is organized as follows: Section 2 presents some background information about ART and the FSCS method. Section 3 introduces our proposed FSCS-CTSR method. Section 4 describes the experimental set-up in detail. Section 5 explains the experimental results, including analyzing the effectiveness and efficiency of FSCS-CTSR. Section 6 discusses potential threats to the validity of our study. Section 7 examines some related work, and Section 8 concludes the paper.

2. Background

2.1. Adaptive random testing

Faulty programs may often have multiple inputs, called *failure-causing inputs*, that cause unexpected outcomes when the program is executed with them. Previous research [11,31] has found that failure-causing inputs tend to cluster into contiguous regions (called *failure regions*) [14]. Generally speaking, the failure regions can be described by two basic characteristics: the *failure rate* and the *failure pattern* [32]. The failure rate is defined as the proportion of failure-causing inputs to all possible inputs. The failure pattern is the location and geometrical shape of the failure-causing inputs, and can be roughly classified into one of three types: *block pattern*, *strip pattern* and *point pattern* [33]. Fig. 1 illustrates these three failure patterns in a 2-dimensional input domain, with the borders representing the input domain boundaries, and the black portion inside each box representing the failure-causing inputs. The block pattern is typically characterized by a cluster of failure-causing inputs that is, more or less, of equal magnitude in each dimension. The strip pattern is similar to a block pattern, but usually with a disproportionate magnitude in at least one dimension. The point pattern comprises failure-causing inputs scattered across the entire input domain, with little or no clustering. Chen et al. [33] reported that block and strip patterns are more commonly found in practical software testing than point patterns. Thus, given a test case that has not revealed a failure, selecting the next test case to be further away may be more likely to reveal a failure than selecting one close by: Well-spread, or *diverse*, test cases should be more effective at finding failures. ART, inspired by this insight, should be a feasible way to enhance random testing.

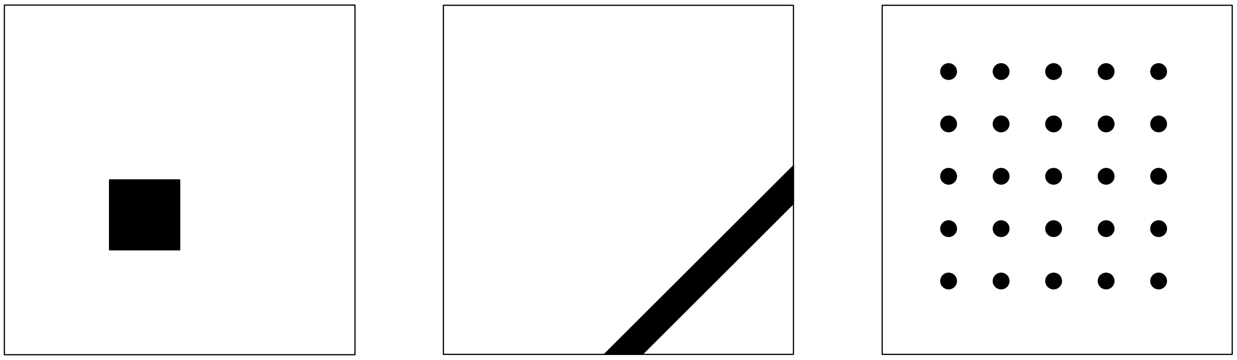

Fig. 1. An illustrative example of three failure pattern in 2-dimensional space.

Algorithm 1 FSCS.

```
1: Set $E = \{\}$; /* executed set */
2: Set $C = \{\}, s$; /* candidate set with size of $s$ */
3: Randomly select a test case $t$ in the input domain;
4: Execute the SUT with $t$;
5: while (termination condition is not met)
6:    Insert $t$ into $E$ as an executed test case;
7:    Randomly select test cases from the input domain, and insert them into $C$ until $C$ has $s$ elements;
8:    Set $Max\_Mindist = Max\_Integer$; /* $Max\_Integer$ is defined as a large numeric value */
9:    Set $c_{best} = null$;
10:   for each candidate $c_i$ $(i = 1, 2, \cdots, s)$, where $c_i \in C$ do
11:      Set $Mindist = Max\_Integer$;
12:      for each executed test case $e \in E$ do
13:         Calculate $dist(c_i, e)$;
14:         if $dist(c_i, e) < Mindist$ then
15:            $Mindist = dist(c_i, e)$;
16:         end_if
17:      end_for
18:      if $Max\_Mindist < Mindist$ then
19:         $Max\_Mindist = Mindist$;
20:         $c_{best} = c_i$;
21:      end_if
22:   end_for
23:   $t = c_{best}$;
24:   Execute the SUT with $t$;
25: end_while
26: return $t$;
```

2.2. *Fixed-size-candidate-set adaptive random testing*

Among the many different ART methods, the Fixed-Size-Candidate-Set ART (FSCS) [34] has been regarded as one of the most popular approaches. FSCS typically uses the Euclidean distance when generating test cases [13], attempting to increase the overall diversity of all test cases by selecting a next test case from a set of candidates such that it has the greatest dissimilarity (distance) from the previously-executed tests [35]. The entire FSCS process is shown in Algorithm 1. FSCS uses an *executed set* (denoted E) containing test cases that have already been executed, but without revealing any failure; and a fixed-size *candidate set* (denoted $C = \{c_1, c_2, \cdots, c_s\}$, with s being the candidate set size). Initially, both E and C are empty. The first test case is randomly selected from the input domain and executed. If this test case does not reveal a failure, it is added to E. Then, s test cases are randomly generated and stored in C, as candidates for the next selection. To identify the best candidate, the distance from each one to its nearest neighbor from the executed test cases is calculated, and then the candidate with the largest distance is selected:

$$\min_{i=1}^{|E|} dist(c_{best}, tc_i) \geq \min_{i=1}^{|E|} dist(c_k, tc_i), \forall c_k \in C \tag{1}$$

where each tc_i $(i \in \{1, 2, \cdots, |E|\})$ is an element of E; c_{best} and each c_k $(k \in \{1, 2, \cdots, s\})$ are elements of C; and $dist$ represents the Euclidean distance between two test cases. This process is repeated until the termination condition (such as finding a failure) is met.

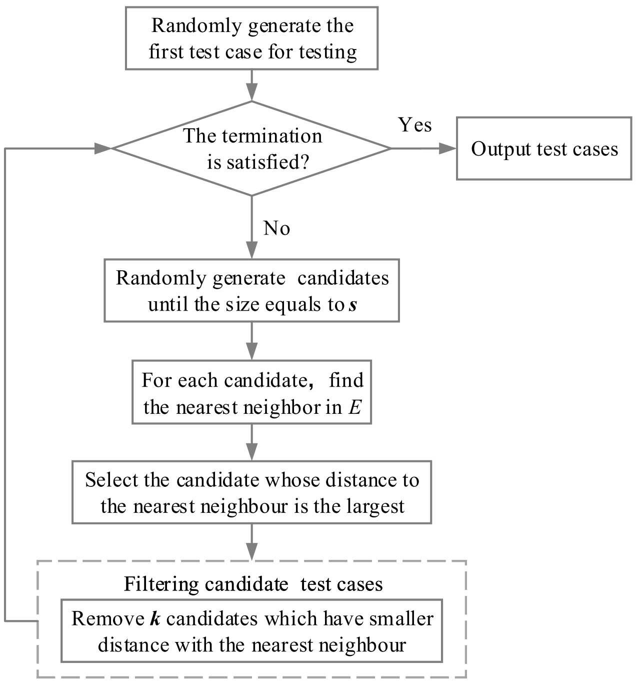
Randomly generate the first test case for testing
The termination is satisfied?
Yes
Output test cases
No
Randomly generate candidates until the size equals to *s*
For each candidate, find the nearest neighbor in *E*
Select the candidate whose distance to the nearest neighbour is the largest
Filtering candidate test cases
Remove *k* candidates which have smaller distance with the nearest neighbour

Fig. 2. FSCS-CTSR flow chart.

Obviously, this process should take more time compared with RT. For a candidate set of size s, dimension d, and n executed test cases, FSCS has a time complexity of

$$O(FSCS) = \sum_{j=1}^{n-1} jds = O(sdn^2) \tag{2}$$

This is a quadratic cost, which will be a very heavy computational overhead when many test cases need to be generated. When the relative test case execution time is short, then the FSCS testing efficiency can be considered to be much lower than RT. For this reason, in spite of its excellent failure-detection capability [29], the practical applications of FSCS have been limited.

3. FSCS by candidate test set reduction

3.1. Framework

Fixed-Size-Candidate-Set Adaptive Random Testing (FSCS) aims to reveal failures using fewer test case executions than RT. By using information about the executed test cases, the overall test case diversity can be significantly enhanced, which can lead to greater testing effectiveness. However, FSCS incurs some heavy computational overheads. To alleviate some of the overheads associated with FSCS, we introduce a new reduction strategy, FSCS by Candidate Test Set Reduction (*FSCS-CTSR*), that reduces the number of new candidate test cases used, thus reducing the total amount of distance calculations. An overview of FSCS-CTSR is shown in Fig. 2. The first test case is randomly selected from the input domain, and used for testing. The testing process continues until the termination condition has been satisfied. In each round after the first test, the next test case is selected from a fixed-size candidate set (of size s). The candidate set has two types of elements: retained test cases, that were generated in a previous round and carried forward; and randomly-generated test cases, that are newly generated in this round. The candidate with the largest minimum distance to previously executed test cases is selected as the next test case to be used. Of the remaining $s - 1$ candidates, k “high quality” (most dissimilar to previous tests) ones are retained; the others are discarded. Later testing rounds will augment the retained, high-quality tests with newly-generated tests to ensure that the candidate set has the full s elements.

3.2. FSCS-CTSR implementation

As explained in the previous section, the number of candidates (s) has a direct influence on the computational overheads of FSCS — the final computational cost is the sum of the individual costs for each candidate. A reduction in the number of

Algorithm 2 FSCS-CTSR.

```
1: Set $E = \{\}$; /* executed set */
2: Set $C = \{\}, s, k$; /* candidate set with size of $s$, $k$ is the number of carried-forward test cases*/
3: Randomly select a test case $t$ in the input domain;
4: Randomly select $k$ test cases in the input domain and insert them into $C$; /* in the first iteration, the carried-forward test cases are instead of random
   test cases */
5: for each candidate $c_i (i = 1, 2, \cdots, k)$, where $c_i \in C$ do
6:    $c_i\_mindist = dist(c_i, t)$; /* $c_i\_mindist$ refers to the minimum distance between $c_i$ and all the executed test cases in $E$ */
7: end_for
8: Execute the SUT with $t$;
9: while (termination condition is not met)
10:    Insert $t$ into $E$ as an executed test case;
11:    for each candidate $c_i (i = k+1, k+2, \cdots, s)$ in $C$ do
12:       Set $Mindist = Max\_Integer$;
13:       for each executed test case $e \in E$ do
14:          Calculate $dist(c_i, e)$;
15:          if $dist(c_i, e) < Mindist$ then
16:             $Mindist = dist(c_i, e)$;
17:          end_if
18:       end_for
19:       $c_i\_mindist = Mindist$;
20:    end_for
21:    select the candidate with the largest minimum distance as $t$, and remove $t$ out of $C$;
22:    for each candidate $c_i (i = 1, 2, \cdots, s-1)$ in $C$ do
23:       Calculate $dist(c_i, t)$;
24:       if $dist(c_i, t) < c_i\_mindist$ then
25:          $c_i\_mindist = dist(c_i, t)$;
26:       end_if
27:    end_for
28:    sort set $C$ and remove $(s-1)-k$ candidates in ascending order of minimum distance;
29:    Execute the SUT with $t$;
30: end_while
31: return $t$;
```

candidates (by, for example, reducing the candidate set size) should, therefore, reduce the overheads. However, the candidate set size also impacts on another essential testing evaluation indicator: Effectiveness. Empirical studies [34] examining the relationship between the candidate set size and testing effectiveness have clearly found the effectiveness to be positively correlated to the size. When choosing the next test case, FSCS selects the test case from the candidate set that has the largest minimum distance to the executed tests. Obviously, the more candidates that there are, the higher will be the possibility of getting a better test case with high "quality", and thus a greater effectiveness. Through experimental studies, Chen et al. [13] found that the effectiveness of FSCS only slightly improved when the size of candidates exceeded 10, so the cost-efficient size was ultimately determined to be 10.

Mak [36] proposed a *Growing Candidate Set ART* (GCS-ART) method that retains the "unused" candidates after each test case selection and execution. The candidate set thus grows in size as the testing process, potentially resulting in a very large number of candidates. However, because some of the candidates that remain in the set into later testing may be of "low quality", the GCS-ART efficiency decreased.

In contrast, FSCS-CTSR aims to alleviate the FSCS computational overheads by reducing the number of candidate test cases, but this could result in a decline in testing effectiveness. To address this, FSCS-CTSR attempts to use some additional candidates that could enhance the effectiveness, but with less cost.

It is interesting to note that FSCS only selects one of the 10 candidates for testing, discarding the rest. According to our observations, after selection, there may often be "high quality" discarded candidates that are only slightly less dissimilar than the best one. These high quality candidates may present better testing performance than the subsequently randomly-generated test cases. More importantly, the distances between these discarded candidates and all the executed test cases have already been calculated — only the distance to the new test case (the best candidate in this iteration) is missing. Calculation of the minimum distances between the best and discarded candidates involves a time complexity of just $O(1)$, compared with the cost of using newly generated candidates of $O(n)$. Therefore, retaining and reusing these discarded (residual) candidates represents an attractive approach to improve the testing effectiveness, with only minimal additional cost. However, some residual candidates may be very similar (close) to the executed test cases, and thus would be unlikely to be selected for testing in later iterations. To further enhance the effectiveness, therefore, we reduced the residual candidates, identifying suitable ones to be retained and used later as additional candidates.

The FSCS-CTSR process is illustrated in Algorithm 2. To better illustrate the FSCS-CTSR steps, we next describe the example shown in Fig. 3. Initially, an empty executed set E and a fixed-size candidate set C are created (in this example, the size is set to 5) (Step 1). The first test case t_1 is randomly selected from the input domain and executed (Fig. 3(a)). If the preset termination condition is met, such as a failure being revealed, then testing is suspended; otherwise, t_1 is added into set E (Step 2). To select the second test case, five candidates $c_1, \cdots, c_5$ are randomly generated and stored in C (Fig. 3(b)). For each candidate, its minimum distance to the executed test cases is calculated, and stored as its similarity

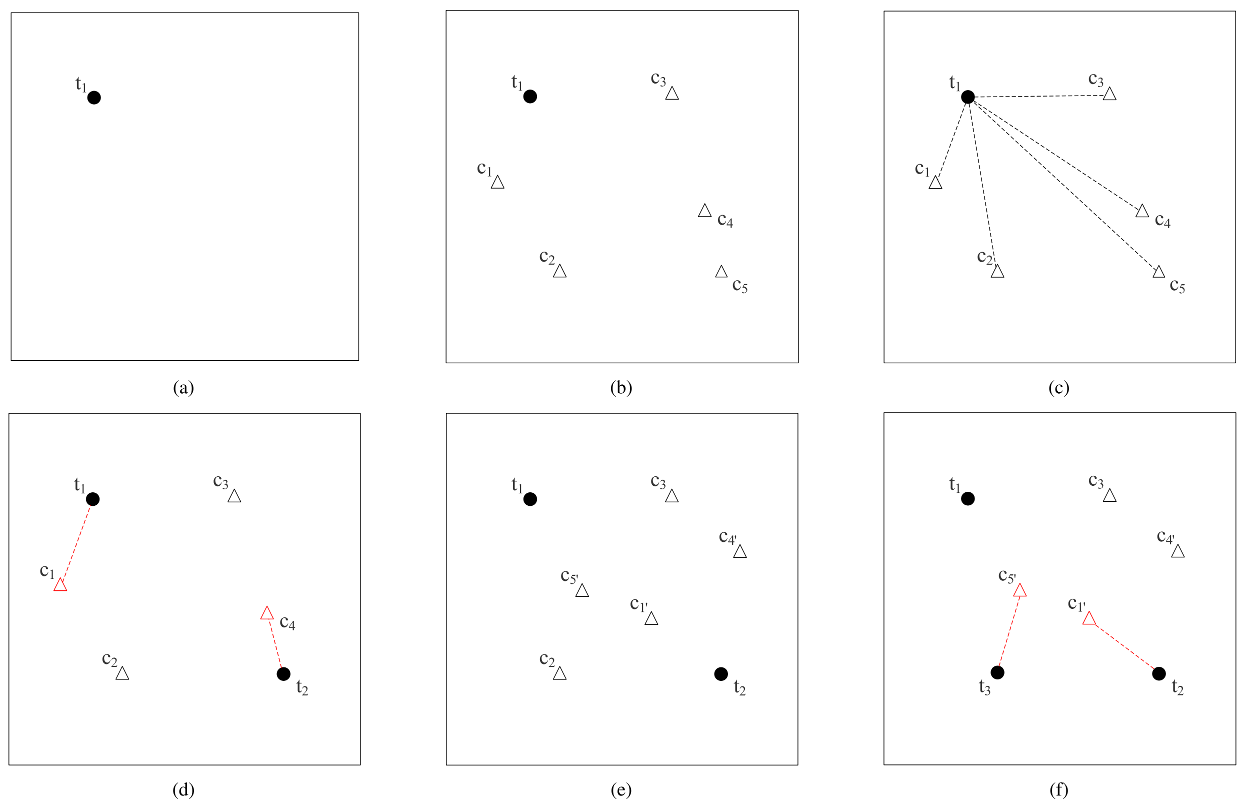

Fig. 3. An illustrative example of FSCS-CTSR in 2-dimensional space.

metric, as shown in Fig. 3(c) (Step 3). The best candidate (the one with the largest minimum distance), c_5, is selected for testing (Fig. 3(d)). If c_5 does not reveal a failure, then it is added into the executed set as t_2, and testing continues (Step 4). (The steps so far are identical to FSCS.) At this point, there are four remaining candidates, all of which would normally now be discarded. To decide which ones are suitable for retention, the minimum distance is examined, and a number, k, are retained: For convenience of illustration, k is set to 2 in this example. (The value will be discussed in detail below.) The two candidates c_2 and c_3 have the largest minimum distances, and are thus carried forward; the rest are discarded (Step 5). Since C now already has two test cases, only three additional candidates need to be generated (Fig. 3(e)). For each carried-forward candidate, the (dis)similarity measurement requires only that the distance from c_5 be calculated and compared with the candidate's previous minimum distance: Its final minimum distance is the smaller of the two (Step 6). For the three new (randomly generated) candidates, the distances to all executed test cases need to be calculated to obtain the minimum distance (Step 7). Then the candidate in C with the largest minimum distance (the best candidate) is selected as the next test case (Fig. 3(f)) (Step 8). Steps 4 to 8 are repeated until the termination condition is satisfied (a failure is revealed).

We conducted an empirical study to determine the optimal value of k (the number of test cases to be retained and carried forward). For a candidate set size of 10, $10 - k$ test cases thus need to be generated in each iteration. Because the computational overhead involved in carrying forward a test case is much less than that associated with randomly generating a candidate, larger values of k will mean lower costs. However, smaller numbers of new candidates may result in a lower possibility of getting "high quality" test cases, which would reduce the testing effectiveness. Because our empirical study showed the FSCS-CTSR improvement in effectiveness to be very small when k was less than 5, we finally set the value of k to be 5.

3.3. *Complexity analysis of FSCS-CTSR*

This section reports on an investigation of the complexity of FSCS-CTSR through a formal mathematical analysis of both the space complexity and the time complexity.

As shown in Algorithms 1 and 2, similar to the original FSCS, FSCS-CTSR needs to save all the executed test cases, a process incurring a space cost of $O(n)$ (where n is the number of executed test cases). An additional overhead of $O(k)$ is caused by storing the "high quality" candidates (to be carried forward) and their minimum distances every iteration (where k is the number of saved candidates). Although FSCS-CTSR carries forward some candidates in each iteration, the final number of stored test cases is always k, and thus the extra space cost is $O(k)$. In summary, the order of space complexity for FSCS-CTSR is $O(n)$.

As explained in Section 2.2, the time complexity of FSCS is $O(sdn^2)$. Because FSCS-CTSR reduces the number of randomly generated candidates, it reduces their associated distance calculations. Calculation of minimum distances between the k stored (carried-forward) candidate test cases and executed test cases only incurs a time overhead of $O(n)$, while calculations for the $s-k$ randomly-generated candidates incur the same cost as the original FSCS, $O(n^2)$. This means that the time complexity for FSCS-CTSR is $O(\frac{sdn^2}{s-k}+n)$.

3.4. Summary of FSCS-CTSR

The efficiency improvement of FSCS-CTSR over FSCS depends only on the number of carried-forward candidates, and is $\frac{s-k}{s}$ times that of FSCS (where k is the number of carried-forward candidates, and s is the candidate set size).

4. Experimental studies

We conducted a series of simulations and experimental studies to examine the effectiveness and efficiency of FSCS-CTSR. This section explains the experimental set-up.

4.1. Research questions

FSCS-CTSR aims to reduce the heavy FSCS computational overheads through candidate test case reduction. Because this alters the selection of test cases, there may be an impact on the failure-detection capability. We therefore designed a series of comprehensive comparisons to evaluate the effectiveness and the efficiency of our FSCS-CTSR implementation against the original FSCS method. Our empirical studies aimed to answer the following research questions:

RQ1 Compared with the original FSCS, how effective is FSCS-CTSR?
RQ2 Does FSCS-CTSR reduce the time taken in testing?

4.2. Variables and measures

We used the original FSCS (and RT) as a base-line for comparisons.

The dependent variable for RQ1 is the metric used to compare and evaluate the failure-detection capability of FSCS-CTSR: We used the *F-measure* [34], which is the expected number of test case executions before finding the first failure. Because software testing may often halt when an unexpected result is encountered, to allow for fault localization and debugging, the F-measure is a good metric for testing [34]. The F-measure has often been used in ART effectiveness evaluations, with lower F-measure values indicating better performance [37]. Here, we denote the F-measure of random testing by F_{RT}, and the F-measure of the ART methods by F_{ART}. The F_{RT} used in our comparisons is a theoretical value equal to $1/\theta$, where θ is the failure rate of the SUT. Because a goal underlying the ART methods is to improve on the effectiveness of random testing, we use the *F-ratio* to directly measure the improvement of the ART method over RT [38]:

$$F\text{-}ratio = \frac{F_{ART}}{F_{RT}} \tag{3}$$

The effectiveness of an ART method is equivalent to RT if the F-ratio $= 1.0$; an F-ratio > 1.0 means that RT achieves greater effectiveness; and an F-ratio < 1.0 means that the ART method is the better one. Smaller F-ratio values indicate better effectiveness.

The dependent variable used for RQ2 was the execution time of the testing methods. In the simulations, the time taken to generate a specific number of test cases was recorded and analyzed. In the experiments, in contrast, we used the *F-time* [39], which is the average time taken before detecting the first failure.

4.3. Simulations and empirical studies

To comprehensively evaluate the effectiveness and the efficiency of FSCS-CTSR, we conducted simulations and empirical studies with 19 real-life programs.

4.3.1. Design of simulations

The simulations were conducted in a unit d-dimensional hyper-space input domain (denoted D) — the magnitude of each dimension was 1.0. Each simulation involved one or more failure regions, which can be understood as the set of all failure-causing inputs, simulating real-life faulty programs. The location of the failure regions was unknown before testing. The failure rate (θ) determines the size of the failure regions and the failure pattern refers to their geometric shape — both of these elements were defined in advance. In our simulations, θ was set to $0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002$, and

0.0001. As described in Section 2, three failure patterns were used in the simulations: block, strip, and point patterns. The block pattern simulations involved a single hypercube, with sides of length $\sqrt[d]{D * \theta}$, being created, and randomly located, within the input domain. The strip patterns were simulated by first randomly identifying two points on adjacent boundaries of the input domain, and then varying the width of the region to correspond to the desired θ. The point pattern simulations consisted of 25 non-overlapping failure regions, all the same size (based on the given θ) and shape, randomly located throughout the input domain.

Test cases that were generated within a simulated failure region were considered to be failure-causing inputs.

4.3.2. Design of empirical studies

It can be argued that simulations alone are not sufficient to fully explore the efficiency and effectiveness of FSCS-CTSR. Real-life programs may be affected by more factors than were reflected in the simulations. Furthermore, the simulation test case executions took a similar amount of time, and thus the obtained F-time may refer to the test case generation, not execution, time. Because of these considerations, in addition to the simulations, we also conducted a series of empirical studies involving real-life programs.

12 real-life C++ programs that have been widely used in ART research [38,40] were selected for our study. These programs have been published in Numerical Recipes [41] and ACM's Collected Algorithms [42]. We also examined a further seven high-dimensional programs written in Java: *calDay*, *complex* and *line* [43]; and *pntLinePos*, *pntTrianglePos*, *twoLinePos* and *triangle* [44]. Mutation testing [45] was used to create mutant versions of the programs under study. Six common mutation operators were used: arithmetic operator replacement (AOR); scalar variable replacement (SVR); constant replacement (CR); return statement replacement (RSR); statement deletion (SD); and relational operator replacement (ROR). Detailed information about the real-life programs is given in Table 1, where *Fault Type* refers to the mutation operator used, and *Total Faults* is the total number of faults generated. For a given program under test, a failure was revealed if a test case resulted in different outcomes for the original program and for the mutant version.

4.4. Statistical analysis

The *p-value* and *effect size* [46] were adopted in our statistical analysis. The *Wilcoxon rank-sum test* was used to verify whether or not there were significant differences between the performance of FSCS-CTSR and that of the original FSCS, where (p-values) results of less than 0.05 indicate a significant difference between them. The non-parametric Vargha and Delaney effect size, $\hat{A}_{12}(X, Y)$, was used to indicate the probability of technique X being better than technique Y. To reduce the impact of randomness in the studies, we ran each group of experiments 3000 times and provide the average results.

4.5. Experiment environment

All experiments were run on a machine running Windows 10 Home OS, equipped with a 2.3 GHz i5 CPU, and 8-GB RAM. Microsoft Visual Studio 2013's Visual C++ was used for coding the C++ programs, and Java 1.8.0 was used to implement the Java programs (*calDay, complex, pntLinePos, triangle, line, pntTrianglePos, twoLinePos*) and our algorithms.[1] All testing was conducted through Java, with the executable C++ programs being called externally, through the Java native interface.

5. Experimental results

Using the results of both the simulations and the experiments, this section answers the research questions posed in Section 4.1.

5.1. Answer to RQ1: testing effectiveness

The F-ratio results for the simulation effectiveness comparisons between FSCS and FSCS-CTSR, for block, strip, and point patterns are presented in Tables 2 to 4. The results are grouped according to input dimensionality, d; the failure rate is represented by θ; the significant differences are indicated by the p-value; and the effect size is denoted by $\hat{A}_{12}$. The F-ratio results for the effectiveness comparisons of the 19 subject programs are reported in Table 5.

5.1.1. Results of simulations

Based on the simulation results reported in Tables 2 to 4, we have the following observations:

1) Similar to FSCS, the effectiveness of FSCS-CTSR decreases as the dimensionality increases.
2) In any given dimension d, the F-ratios for FSCS-CTSR and FSCS decrease as the failure rate decreases, for block and point patterns: Both show better effectiveness for lower failure rates.

[1] The source code for FSCS-CTSR and the experiments, and relevant experimental data, is available at: https://github.com/huangrubing/FSCS-CTSR.

Table 1
Subject programs for empirical studies.

Program	Dimension	Input	Input Domain		Size	Fault Types	Total	Failure Rate
	(d)	Type	From	To	(LOC)		Faults	(θ)
airy	1	float	-5000	5000	43	CR	1	0.000716
bessj0	1	float	-300000	300000	28	AOR, ROR, SVR, CR	5	0.001373
erfcc	1	float	-30000	30000	14	AOR, ROR, SVR, CR	4	0.000574
probks	1	float	-50000	50000	22	AOR, ROR, SVR, CR	4	0.000387
tanh	1	float	-500	500	18	AOR, ROR, SVR, CR	4	0.001817
bessj	2	float	(2, -1000)	(300, 15000)	99	AOR, ROR, CR	4	0.001298
gammq	2	float	(0, 0)	(1700, 40)	106	ROR, CR	4	0.000830
sncndn	2	float	(-5000, -5000)	(5000, 5000)	64	SVR, CR	5	0.001623
golden	3	float	(-100, -100, -100)	(60, 60, 60)	80	ROR, SVR, CR	5	0.000550
plgndr	3	float, int	(10, 0, 0)	(500, 11, 1)	36	AOR, ROR, CR	5	0.000368
cel	4	float	(0.001, 0.001, 0.001, 0.001)	(1, 300, 10000, 1000)	49	AOR, ROR, CR	3	0.000332
el2	4	float	(0, 0, 0, 0)	(250, 250, 250, 250)	78	AOR, ROR, SVR, CR	9	0.000690
calDay	5	int	(1, 1, 1, 1, 1800)	(12, 31, 12, 31, 2200)	37	SD	1	0.000632
complex	6	int	(-20, -20, -20, -20, -20, -20)	(20, 20, 20, 20, 20, 20)	68	SVR	1	0.000901
pntLinePos	6	float	(-25, -25, -25, -25, -25, -25)	(25, 25, 25, 25, 25, 25)	23	CR	1	0.000728
triangle	6	int	(-25, -25, -25, -25, -25, -25)	(25, 25, 25, 25, 25, 25)	21	CR	1	0.000713
line	8	int	(-10, -10, -10, -10, -10, -10, -10, -10)	(10, 10, 10, 10, 10, 10, 10, 10)	86	ROR	1	0.000303
pntTrianglePos	8	float	(-10, -10, -10, -10, -10, -10, -10, -10)	(10, 10, 10, 10, 10, 10, 10, 10)	68	CR	1	0.000141
twoLinePos	8	float	(-15, -15, -15, -15, -15, -15, -15, -15)	(15, 15, 15, 15, 15, 15, 15, 15)	28	CR	1	0.000133

Table 2
F-ratio simulation comparisons between FSCS-CTSR and FSCS, for block patterns.

Dimension (d)	Failure Rate (θ)	F-ratio				Statistical Analysis	
		FSCS-ART (x)	FSCS-CTSR (y)	Difference ($x-y$)	Improvement ($1-y/x$)	p-value	$\hat{A}_{12}$
$d=1$	0.01	0.5668	0.5676	-0.0008	-0.14%	0.59	0.50
	0.005	0.5532	0.5721	-0.0189	-3.42%	0.40	0.49
	0.002	0.5673	0.5620	0.0053	0.93%	0.46	0.51
	0.001	0.5596	0.5649	-0.0053	-0.94%	0.90	0.50
	0.0005	0.5705	0.5716	-0.0011	-0.19%	0.37	0.51
	0.0002	0.5648	0.5712	-0.0064	-1.13%	0.68	0.50
	0.0001	0.5680	0.5650	0.0030	0.52%	0.73	0.50
$d=2$	0.01	0.6707	0.6690	0.0017	0.25%	0.53	0.50
	0.005	0.6577	0.6552	0.0026	0.39%	0.54	0.50
	0.002	0.6475	0.6539	-0.0064	-1.00%	0.84	0.50
	0.001	0.6342	0.6386	-0.0044	-0.69%	0.33	0.51
	0.0005	0.6446	0.6364	0.0083	1.28%	0.20	0.51
	0.0002	0.6367	0.6255	0.0112	1.76%	0.08	0.51
	0.0001	0.6105	0.6401	-0.0296	-4.85%	0.17	0.49
$d=3$	0.01	0.8319	0.8171	0.0148	1.78%	0.53	0.50
	0.005	0.8046	0.7921	0.0126	1.56%	0.44	0.51
	0.002	0.7763	0.7701	0.0062	0.80%	0.88	0.50
	0.001	0.7649	0.7457	0.0192	2.51%	0.15	0.51
	0.0005	0.7412	0.7453	-0.0041	-0.55%	0.90	0.50
	0.0002	0.7323	0.7285	0.0038	0.52%	0.27	0.51
	0.0001	0.7258	0.7138	0.0120	1.65%	0.30	0.51
$d=4$	0.01	1.0517	1.0315	0.0203	1.93%	0.42	0.51
	0.005	0.9902	0.9827	0.0075	0.76%	0.76	0.50
	0.002	0.9159	0.9240	-0.0081	-0.88%	0.67	0.50
	0.001	0.8847	0.8811	0.0036	0.41%	0.85	0.50
	0.0005	0.8722	0.8487	0.0234	2.69%	0.13	0.51
	0.0002	0.8495	0.8362	0.0133	1.57%	0.57	0.50
	0.0001	0.8172	0.8346	-0.0174	-2.13%	0.54	0.50
$d=5$	0.01	1.3083	1.2882	0.0201	1.54%	0.53	0.50
	0.005	1.2137	1.1603	0.0534	4.40%	0.03	0.52
	0.002	1.1421	1.0951	0.0470	4.11%	0.02	0.52
	0.001	1.1095	1.0619	0.0476	4.29%	0.08	0.51
	0.0005	1.0706	1.0135	0.0571	5.33%	0.04	0.51
	0.0002	1.0118	1.0029	0.0089	0.88%	0.45	0.51
	0.0001	0.9533	0.9882	-0.0349	-3.67%	0.10	0.49
$d=8$	0.01	2.5275	2.4333	0.0942	3.73%	0.26	0.51
	0.005	2.3643	2.2291	0.1352	5.72%	0.05	0.51
	0.002	2.1339	1.9714	0.1626	7.62%	0.01	0.52
	0.001	1.9690	1.8442	0.1249	6.34%	0.02	0.52
	0.0005	1.8483	1.6766	0.1717	9.29%	0.00	0.53
	0.0002	1.6653	1.5929	0.0724	4.35%	0.07	0.51
	0.0001	1.5854	1.4725	0.1130	7.13%	0.03	0.52

3) For block patterns, FSCS-CTSR has comparable failure-detection performance to FSCS in low-dimensional space. According to the statistical analysis, when $d \leq 4$, almost all the p-values are greater than 0.05, and all the effect sizes are close to 0.5, which indicates that there is no significant difference between FSCS-CTSR and FSCS. However, as the number of dimensions increases beyond $d = 4$, FSCS-CTSR shows better effectiveness.
4) For strip patterns, the effectiveness of both methods are quite similar, regardless of dimensionality and failure rate. This is supported by the statistical analysis, where most of the p-values are greater than 0.05, and almost all the effect sizes are very close to (within 0.01 of) 0.5. All the algorithms show slightly greater effectiveness than RT, except when $d = 1$.
5) For point patterns, the F-ratios of the two methods rise as the number of dimensions increases. Compared with RT, the ART effectiveness is slightly better when $d = 1$ and $d = 2$. However, RT performs better when $d \geq 3$. According to the statistical analysis, there is no significant difference between the FSCS-CTSR and FSCS when $d \leq 4$. However, for $d = 5$ and $d = 8$, FSCS-CTSR has better testing effectiveness.

Observation 1 is as expected. Chen et al. [35] noted that high-dimensional input domains have more boundaries, with candidates near the boundaries being more likely to be selected as the optimal test case. This results in many selected test cases often concentrating near the boundary, thereby violating the notation of "evenly spread", and performing less well: Both the original FSCS and FSCS-CTSR show larger F-ratios in higher dimensions.

Table 3
F-ratio simulation comparisons between FSCS-CTSR and FSCS, for strip patterns.

Dimension (d)	Failure Rate (θ)	F-ratio				Statistical Analysis	
		FSCS-ART (x)	FSCS-CTSR (y)	Difference ($x-y$)	Improvement ($1-y/x$)	p-value	$\hat{A}_{12}$
$d=1$	0.01	0.5705	0.5649	0.0056	0.99%	0.64	0.50
	0.005	0.5690	0.5603	0.0087	1.53%	0.28	0.51
	0.002	0.5627	0.5664	-0.0037	-0.65%	0.95	0.50
	0.001	0.5672	0.5654	0.0018	0.31%	0.25	0.51
	0.0005	0.5703	0.5799	-0.0096	-1.69%	0.92	0.50
	0.0002	0.5577	0.5682	-0.0105	-1.89%	0.84	0.50
	0.0001	0.5545	0.5776	-0.0230	-4.16%	0.19	0.49
$d=2$	0.01	0.9042	0.9366	-0.0324	-3.59%	0.14	0.49
	0.005	0.9446	0.9441	0.0004	0.05%	0.59	0.50
	0.002	0.9596	0.9273	0.0324	3.37%	0.52	0.50
	0.001	1.0008	0.9632	0.0377	3.76%	0.57	0.50
	0.0005	0.9529	1.0192	-0.0663	-6.96%	0.00	0.48
	0.0002	0.9965	1.0178	-0.0213	-2.14%	0.48	0.49
	0.0001	0.9736	1.0147	-0.0411	-4.22%	0.21	0.49
$d=3$	0.01	0.9679	0.9726	-0.0048	-0.49%	0.61	0.50
	0.005	0.9932	1.0021	-0.0089	-0.90%	0.77	0.50
	0.002	0.9802	0.9887	-0.0084	-0.86%	0.46	0.49
	0.001	0.9805	0.9775	0.0030	0.31%	0.49	0.49
	0.0005	0.9562	0.9934	-0.0372	-3.89%	0.09	0.49
	0.0002	1.0133	1.0316	-0.0183	-1.80%	0.91	0.50
	0.0001	0.9750	0.9944	-0.0194	-1.99%	0.30	0.49
$d=4$	0.01	0.9880	0.9672	0.0208	2.10%	0.97	0.50
	0.005	0.9775	0.9785	-0.0010	-0.10%	0.68	0.50
	0.002	1.0043	0.9794	0.0248	2.47%	0.30	0.51
	0.001	1.0136	1.0020	0.0115	1.14%	0.98	0.50
	0.0005	0.9808	1.0128	-0.0320	-3.26%	0.54	0.50
	0.0002	0.9643	0.9828	-0.0185	-1.92%	0.78	0.50
	0.0001	1.0074	0.9900	0.0174	1.73%	0.60	0.50
$d=5$	0.01	0.9753	1.0205	-0.0452	-4.63%	0.21	0.49
	0.005	0.9816	0.9585	0.0231	2.35%	0.42	0.51
	0.002	1.0085	0.9703	0.0382	3.79%	0.06	0.51
	0.001	1.0046	0.9818	0.0228	2.26%	0.25	0.51
	0.0005	0.9958	0.9889	0.0069	0.69%	0.36	0.51
	0.0002	1.0124	1.0238	-0.0114	-1.13%	0.39	0.49
	0.0001	0.9890	0.9896	-0.0006	-0.06%	0.66	0.50
$d=8$	0.01	0.9843	0.9611	0.0232	2.36%	0.36	0.51
	0.005	1.0300	0.9888	0.0412	4.00%	0.10	0.51
	0.002	1.0111	0.9747	0.0364	3.60%	0.26	0.51
	0.001	1.0288	1.0168	0.0120	1.17%	0.98	0.50
	0.0005	0.9982	1.0025	-0.0043	-0.43%	0.96	0.50
	0.0002	0.9863	0.9823	0.0040	0.41%	0.76	0.50
	0.0001	0.9849	0.9823	0.0026	0.26%	0.69	0.50

Observation 2 can be explained by the fact that “evenly spread” distributions of test cases are more often reached after a larger number of test cases have been executed. Both methods show better performance for lower failure rates.

Observations 3–5, regarding comparable performance for the three different failure pattern types, are as expected. Although our method alters the FSCS test case selection, the carried-forward candidates ensure the “quality” of the candidate set. Because the quality of our method’s candidate set is similar to that of FSCS in low-dimensional space, the effectiveness of both methods is also similar. Also, as the dimensions increase, both FSCS and FSCS-CTSR suffer from the *curse of dimensionality* [47], with a sharp increase in the F-ratio. Partly due to the carried-forward candidates being less impacted by the curse of dimensionality than FSCS candidates, FSCS-CTSR shows better effectiveness when $d>4$.

In summary, when the dimensionality is low ($d \leq 4$), FSCS-CTSR has comparable effectiveness to FSCS. However, as the dimensionality increases beyond $d=4$, FSCS-CTSR obtains better failure-detection compared with FSCS.

5.1.2. Results of empirical studies

The experimental results from applying FSCS and FSCS-CTSR to the 19 real-life programs are presented in Table 5. Based on these results, we have the following observations:

Table 4
F-ratio simulation comparisons between FSCS-CTSR and FSCS, for point patterns.

Dimension (d)	Failure Rate (θ)	F-ratio				Statistical Analysis	
		FSCS-ART (x)	FSCS-CTSR (y)	Difference ($x-y$)	Improvement ($1-y/x$)	p-value	$\hat{A}_{12}$
$d=1$	0.01	0.9646	0.9731	-0.0085	-0.88%	0.92	0.50
	0.005	0.9485	0.9631	-0.0147	-1.55%	0.44	0.49
	0.002	0.9520	0.9817	-0.0297	-3.12%	0.63	0.50
	0.001	0.9341	0.9515	-0.0174	-1.86%	0.66	0.50
	0.0005	0.9829	0.9441	0.0388	3.95%	0.09	0.51
	0.0002	0.9449	0.9519	-0.0070	-0.74%	0.41	0.49
	0.0001	0.9174	0.9686	-0.0512	-5.58%	0.09	0.49
$d=2$	0.01	0.9646	0.9731	-0.0085	-0.88%	0.92	0.50
	0.005	0.9485	0.9631	-0.0147	-1.55%	0.44	0.49
	0.002	0.9520	0.9817	-0.0297	-3.12%	0.63	0.50
	0.001	0.9341	0.9515	-0.0174	-1.86%	0.66	0.50
	0.0005	0.9829	0.9441	0.0388	3.95%	0.09	0.51
	0.0002	0.9449	0.9519	-0.0070	-0.74%	0.41	0.49
	0.0001	0.9174	0.9686	-0.0512	-5.58%	0.09	0.49
$d=3$	0.01	1.0965	1.0936	0.0029	0.27%	0.49	0.51
	0.005	1.0740	1.0810	-0.0070	-0.65%	0.87	0.50
	0.002	1.0321	1.0321	0.0000	0.00%	0.52	0.50
	0.001	1.0283	0.9983	0.0300	2.91%	0.33	0.51
	0.0005	1.0332	1.0251	0.0082	0.79%	0.49	0.51
	0.0002	1.0268	0.9726	0.0542	5.28%	0.04	0.52
	0.0001	1.0280	0.9849	0.0431	4.19%	0.27	0.51
$d=4$	0.01	1.2918	1.2540	0.0378	2.93%	0.11	0.51
	0.005	1.2619	1.2073	0.0546	4.32%	0.11	0.51
	0.002	1.1419	1.1289	0.0131	1.15%	0.99	0.50
	0.001	1.1310	1.1122	0.0187	1.66%	0.59	0.50
	0.0005	1.1036	1.1068	-0.0032	-0.29%	0.62	0.50
	0.0002	1.0674	1.0461	0.0213	1.99%	0.43	0.51
	0.0001	1.0582	1.0562	0.0019	0.18%	0.94	0.50
$d=5$	0.01	1.5700	1.5048	0.0652	4.16%	0.18	0.51
	0.005	1.4708	1.3926	0.0782	5.32%	0.00	0.52
	0.002	1.3426	1.3032	0.0394	2.93%	0.23	0.51
	0.001	1.2740	1.2475	0.0265	2.08%	0.11	0.51
	0.0005	1.2700	1.2420	0.0280	2.20%	0.30	0.51
	0.0002	1.2081	1.1822	0.0259	2.14%	0.37	0.51
	0.0001	1.1634	1.1467	0.0168	1.44%	0.53	0.50
$d=8$	0.01	2.7005	2.5830	0.1175	4.35%	0.18	0.51
	0.005	2.4977	2.3336	0.1641	6.57%	0.02	0.52
	0.002	2.2672	2.1032	0.1640	7.23%	0.00	0.52
	0.001	2.1447	1.9756	0.1691	7.89%	0.00	0.53
	0.0005	1.9239	1.8262	0.0977	5.08%	0.03	0.52
	0.0002	1.7996	1.6958	0.1038	5.77%	0.05	0.51
	0.0001	1.7267	1.6321	0.0946	5.48%	0.02	0.52

1) Compared with RT, both methods have better failure-detection capability in low-dimensional space ($d \leq 5$), except for the sncndn and golden programs. In higher dimensions ($d \geq 6$), the effectiveness of the two methods appears relatively poor, except for with the pntTrianglePos program.
2) Compared with FSCS, FSCS-CTSR has similar or better effectiveness for 17 of the real-life programs (not for cel or el2). However, almost all the p-values for F-ratio comparisons between FSCS-CTSR and the original FSCS for these 17 programs are much greater than 0.05, indicating that the differences are not significant. Furthermore, almost all the effect sizes are very close to (within 0.01 or 0.02 of) 0.5. This means that it is not possible to categorically say which method is better.

Regarding Observation 1, in most cases, FSCS and FSCS-CTSR outperform RT, particularly in low dimensional space. However, both methods are significantly affected by the curse of dimensionality, showing decreasing failure-detection capability as the dimensionality increases. Generally speaking, anywhere that the original FSCS method may be used, the more efficient FSCS-CTSR method should be considered instead. It is therefore recommended that FSCS-CTSR be used for low-dimensional input domain SUTs, and/or where the test case execution is expensive. If, somehow, the SUT is expected to have a (very) low failure rate, then FSCS-CTSR should be considered.

Observation 2 relates to the very similar performance of FSCS-CTSR and the original FSCS, for most subject programs. Further investigation of cel and el2 revealed that their failure regions were both near the boundaries of the input domains.

Table 5
F-ratio comparisons between FSCS-CTSR and FSCS for the 19 subject programs.

Program	Dimension (d)	F-ratio				Statistical Analysis	
		FSCS-ART (x)	FSCS-CTSR (y)	Difference ($x - y$)	Improvement ($1 - y/x$)	p-value	$\hat{A}_{12}$
airy	1	0.5726	0.5715	0.0011	0.19%	0.69	0.50
bessj0	1	0.6125	0.6072	0.0053	0.86%	0.33	0.51
erfcc	1	0.5964	0.5887	0.0078	1.30%	0.07	0.51
probks	1	0.5629	0.5623	0.0006	0.11%	0.93	0.50
tanh	1	0.5737	0.5661	0.0076	1.33%	0.41	0.51
bessj	2	0.5768	0.5928	-0.0161	-2.79%	0.33	0.49
gammq	2	0.9023	0.8769	0.0254	2.82%	0.49	0.51
sncndn	2	1.0170	1.0179	-0.0009	-0.09%	0.66	0.50
golden	3	1.0299	0.9899	0.0400	3.88%	0.21	0.51
plgndr	3	0.5972	0.6037	-0.0065	-1.09%	0.37	0.49
cel	4	0.5177	0.5732	-0.0555	-10.72%	0.00	0.47
el2	4	0.5052	0.5391	-0.0339	-6.71%	0.01	0.48
calDay	5	0.8177	0.8173	0.0004	0.04%	0.79	0.50
complex	6	1.0014	1.1821	-0.1807	-18.05%	0.32	0.49
pntLinePos	6	1.0482	1.0589	-0.0107	-1.02%	0.99	0.50
triangle	6	0.9905	0.9716	0.0189	1.91%	0.54	0.50
line	8	1.0116	0.9515	0.0602	5.95%	0.01	0.52
pntTrianglePos	8	0.6710	0.6926	-0.0216	-3.22%	0.16	0.49
twoLinePos	8	1.0514	1.0501	0.0013	0.13%	0.65	0.50

As explained earlier, FSCS may have a bias towards selecting test cases near to the input domain boundary, and hence can exhibit better effectiveness for programs with failure regions there.

Overall, FSCS-CTSR has demonstrated similar failure-detection capability to FSCS for the real-life subject programs.

Summary of answers to RQ1: The results from both the simulations and experiments indicate that FSCS-CTSR has comparable, or better, effectiveness compared with the original FSCS.

5.2. Answers to RQ2: testing efficiency

5.2.1. Results of simulations

Fig. 4 presents the efficiency results comparing the test case generation time for FSCS with that for FSCS-CTSR. The X-axis shows the number of executed test cases, and the Y-axis gives their generation time. Based on these results, we have the following observations:

1) Both the original FSCS and FSCS-CTSR show rapidly-increasing time costs as the number of executed test cases increases.
2) FSCS-CTSR shows much better efficiency than FSCS in all cases.
3) The improvement in efficiency of FSCS-CTSR over FSCS remains almost stable, regardless of dimensionality.

Regarding Observation 1, as mentioned earlier, the time complexity of FSCS-CTSR is $O(\frac{sdn^2}{s-k}+n)$, which is still quadratic complexity. Because of this, the time overheads increase sharply when generating more test cases. However, the proposed FSCS-CTSR method involves a new strategy of reusing "high quality" candidates to address the complexity issue. This advance may benefit many algorithms in the ART family: Other efficient ART methods (including KDFC-ART [48] and Mirror-ART [49]) could be further improved using this strategy.

Observation 2 relates to the similarity checking (distance calculations) for candidates: The carried-forward candidates reduce these distance calculation overheads, enabling FSCS-CTSR to outperform FSCS in terms of efficiency.

The stability of improvement (Observation 3) relates to the number of carried-forward candidates (k): Once k is set, the time complexity is determined, which also determines the overall FSCS-CTSR efficiency improvement.

In summary, the simulation results indicate that FSCS-CTSR achieves better efficiency than FSCS.

5.2.2. Results of empirical studies

Table 6 presents the F-time results for FSCS and FSCS-CTSR for the 19 real-life subject programs. According to the p-value and effect size data, it can be concluded that FSCS-CTSR has better efficiency than FSCS, in all cases. It can also be observed that the Table 6 results are consistent with the simulation results. Overall, FSCS-CTSR incurs lower computational overheads than the original FSCS.

Summary of answers to RQ2: According to the results from both the simulations and experiment, FSCS-CTSR requires less time to reveal the first failure in testing than FSCS. In other words, FSCS-CTSR is cost-effective.

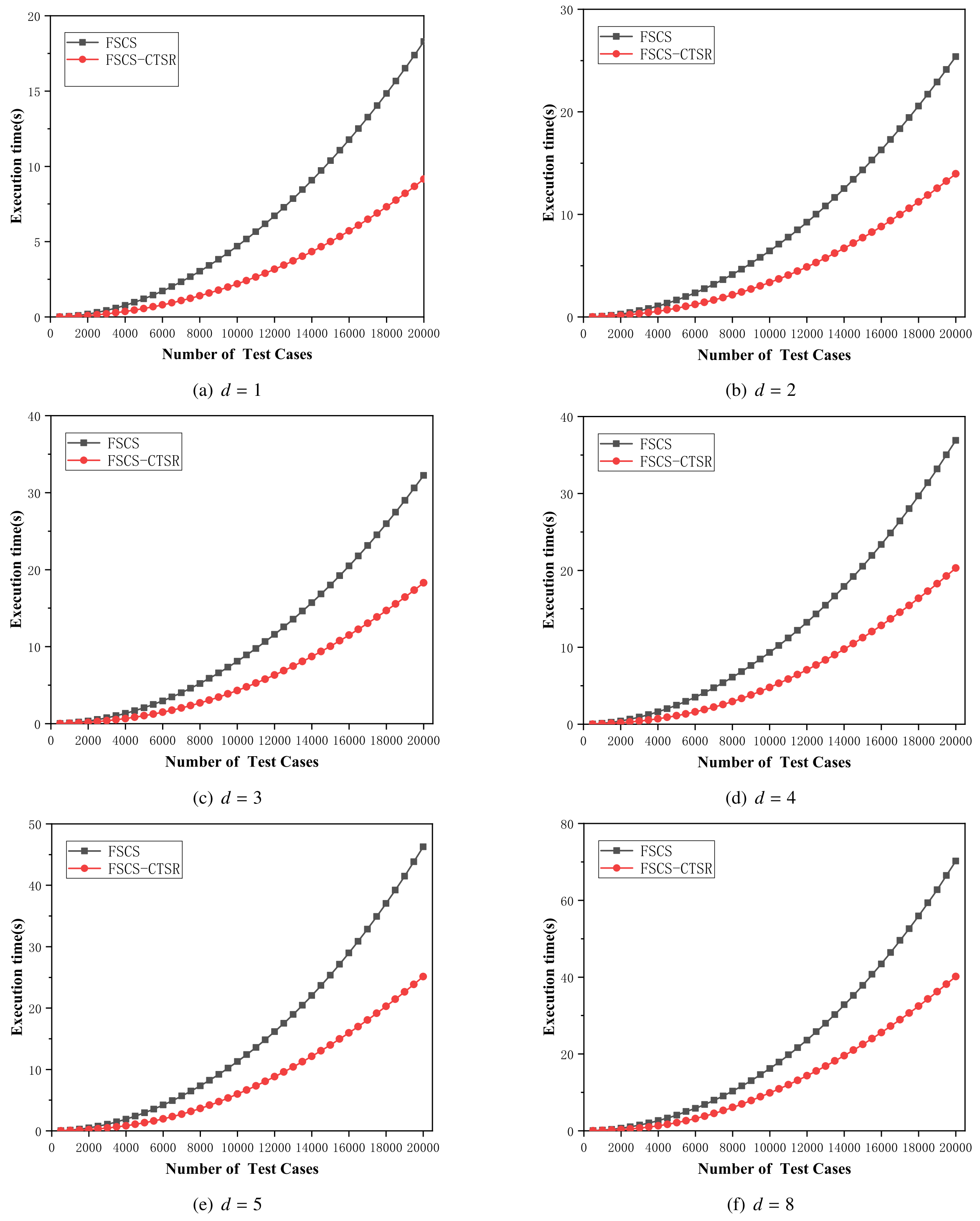

Fig. 4. Test case generation time for FSCS-CTSR and original FSCS in simulations.

6. Threats to validity

In this section, we discuss some potential threats to the validity of our study, including the construct, external, and internal validity.

Table 6
F-time comparisons between FSCS-CTSR and FSCS for 19 subject programs.

Program	Dimension (d)	F-ratio				Statistical Analysis	
		FSCS-ART (x)	FSCS-CTSR (y)	Difference ($x-y$)	Improvement ($1-y/x$)	p-value	$\hat{A}_{12}$
airy	1	35.66	18.00	17.66	49.52%	0.00	0.61
bessj0	1	10.33	5.23	5.10	49.39%	0.00	0.61
erfcc	1	60.58	29.30	31.28	51.64%	0.00	0.63
probks	1	148.42	83.47	64.95	43.76%	0.00	0.59
tanh	1	4.70	2.39	2.31	49.18%	0.00	0.62
bessj	2	19.22	10.79	8.43	43.87%	0.00	0.56
gammq	2	128.18	57.86	70.32	54.86%	0.00	0.59
sncndn	2	42.94	20.72	22.21	51.74%	0.00	0.57
golden	3	474.36	222.94	251.42	53.00%	0.00	0.58
plgndr	3	324.41	162.85	161.56	49.80%	0.00	0.58
cel	4	402.40	221.71	180.69	44.90%	0.00	0.56
el2	4	89.70	47.84	41.86	46.67%	0.00	0.56
calDay	5	337.84	164.85	172.99	51.20%	0.00	0.58
complex	6	253.88	127.43	126.45	49.81%	0.00	0.58
pntLinePos	6	417.61	208.45	209.16	50.08%	0.00	0.59
triangle	6	407.32	188.02	219.31	53.84%	0.00	0.59
line	8	2675.86	1459.65	1216.20	45.45%	0.00	0.61
pntTrianglePos	8	5123.10	2946.45	2176.65	42.49%	0.00	0.58
twoLinePos	8	12694.66	7182.29	5512.37	43.42%	0.00	0.60

6.1. *Threats to construct validity*

Construct validity relates to whether or not the experiment measurements and metrics are consistent with its claims. In this paper, the effectiveness of testing methods in our experiments was examined using the F-measure [37], which is the expected number of test case executions required to reveal the first failure. Both the P-measure and E-measure are also popular evaluation metrics in software testing, and our results may be different if these metrics are used. However, both P-measure and E-measure refer to situations where the number of test cases is fixed. On the contrary, the topic of our research is the incremental generation of test cases by ART. Therefore, the F-measure is a more suitable metric for our analysis.

On the other hand, another potential threat to our study's validity may relate to the sizes of real-life programs that we used: Our programs were relatively small, with only a few mutant faults seeded-in. Our future work will involve larger subject programs, and a larger variety of seeded faults.

6.2. *Threats to external validity*

External validity refers to whether or not the results may be universal, under other conditions. In this paper, we assumed that the set of failure-causing inputs can be classified into one of three regular types [31]: block, strip, and point pattern. However, the actual failure regions may be more complicated in reality than those of the simulations.

Furthermore, our simulations and empirical studies were carried out in numerical input domains: Additional work is needed to extend our method to other, non-numerical input domains.

6.3. *Threats to internal validity*

The internal validity of a study refers to whether or not mistakes were made in the experiments. To reduce the impact of randomness, each group of our experiments was repeated 3000 times. Different parameter settings may also impact the experiments: For example, although we set the number of carried-forward candidates to be 5, the use of other values will impact the testing results. This is something we look forward to exploring further in our future work.

7. Related work

Compared with RT, ART typically has better failure-detection capability, achieved by evenly distributing test cases across the input domain. Various ART methods have been proposed, and many applications implemented. However, the ART processes often incur heavy computational costs. A number of techniques have been proposed to alleviate some of the ART overheads, including: *Mirror Adaptive Random Testing* (MART) [38,49], *Filtering* [50], *Forgetting* [51], *Quasi-Random Testing* (QRT) [52–54], and *FSCS by adopted K-Dimensional tree* (KDFC) [48]. This section briefly introduces these methods.

7.1. *Mirror Adaptive Random Testing (MART)*

Chen et al. [49] proposed MART to reduce the high computational overheads of ART. MART divides the input domain into a number of disjoint, equally-sized subdomains before testing. The main test generation algorithm, e.g. FSCS, is only applied in one subdomain, called the source subdomain. Test cases are then mapped from the source subdomain to the other (mirror) subdomains. Our previous work [38] improved MART efficiency with new mirror functions, and a "divide-and conquer" strategy that incrementally partitions the input domain. In addition, MART methods have the potential risk of the failure region being "shredded" [55] when dividing the input domain, severely reducing the testing effectiveness.

7.2. Adaptive Random Testing with Filtering

Adaptive Random Testing with Filtering [50] was developed to improve the efficiency of the *Restricted Random Testing* (RRT) [18,19] version of ART. The method employs a *bounding region* (a hypercube) around candidate test cases, avoiding distance calculations for those executed test cases outside the region. In contrast to that method, FSCS-CTSR was developed for the FSCS version of ART.

7.3. Forgetting

In general, the computational overhead of ART strongly relates to the number of executed test cases ($|E|$), with more executed test cases meaning higher costs. Chan et al. [51] proposed an overheads reduction method called *forgetting*, which reduced the number of executed test cases to be examined to a (small) fixed number. Once the number of executed test cases reaches the preset fixed number, one of three approaches is followed to maintain the executed set size, reducing the complexity of the forgetting strategy to a linear order. However, forgetting faces the challenge of how to decide the fixed number, with the interplay between this number and the software's failure rate resulting in different effectiveness. FSCS-CTSR can not only achieve better efficiency, but also maintains comparable or better failure-finding effectiveness to FSCS.

7.4. Quasi-Random Testing (QRT)

Quasi-Random Testing (QRT) takes advantage of quasi-random sequences to generate points with low discrepancy, incurring only a linear-order time complexity [52]. Unlike FSCS-CTSR, QRT is a standalone testing method.

7.5. KDFC-ART

Recently, Mao et al. [48] proposed KDFC-ART, a technique that can quickly find the nearest neighbors of candidates, based on spatial indexing. The original KDFC (Native KDFC) divided an input domain into multiple subdomains with respect to each dimension, and constructed a KD-tree. Thus, for each candidate, multiple test cases that were far away could be skipped in the distance computations. To further improve the KD-tree balance in Native-KDFC, SemiBal-KDFC was introduced, demonstrating better efficiency. A challenge to KDFC related to backtracking in the KD-tree: In the worst case, all nodes in the tree may need to be traversed, severely reducing the efficiency. Mao et al. proposed a third version of KDFC, LimBal-KDFC, which kept node traversal under strict control. Both KDFC-ART and FSCS-CTSR aim to improve the testing efficiency by removing unnecessary distance computations. However, the KDFC-ART methods require a KD-tree structure, which results in a high space complexity. When the input domain dimensionality is high, the space cost may become very expensive.

8. Conclusions and future work

Adaptive random testing (ART) is a family of testing techniques that aim to improve the failure-detection capability of RT by more evenly distributing test cases across the input domain. Test cases generated by ART typically have less similarity to each other, and have a higher probability of executing code not previously executed. Previous studies have demonstrated that ART requires fewer test case executions before finding a failure than RT. Among the various ART methods, the *Fixed-Size-Candidate-Set* (FSCS) version of ART is one of the most effective, as shown in many studies [17,26,56]. However, the heavy computational overheads incurred have limited the practical application of FSCS.

In this paper, we have proposed an enhanced FSCS method, *FSCS by Candidate Test Set Reduction* (FSCS-CTSR), which makes use of unselected candidates that have low similarity to the already-executed test cases. According to the results of our simulations and experiments, FSCS-CTSR significantly reduces the FSCS computational overheads while maintaining comparable (or better) failure-finding effectiveness. FSCS-CTSR has comparable effectiveness in low-dimensional space, but better effectiveness in high dimensions.

The FSCS-CTSR strategy is independent of ART methods, which means that it can be applied to other cost-effectiveness algorithms, such as KDFC, to further improve their efficiency. Although FSCS-CTSR is cost-effective, and maintains the FSCS failure-detection effectiveness, it still faces some challenges, such as the rapidly-decreasing efficiency when the number of executed test cases increases. Our future work will include exploring how to achieve a linear-time-complexity FSCS method.

CRediT authorship contribution statement

Rubing Huang: Conceptualization, Methodology, Writing – reviewing & editing, Investigation.
Haibo Chen: Software, Data curation, Writing – original draft reparation.
Weifeng Sun: Visualization, Formal analysis, Methodology verification.
Dave Towey: Validation, Writing – reviewing & editing, Proof-reading.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work is partly supported by the National Natural Science Foundation of China, under grant nos. 61872167 and 61502205. This work is also in part supported by the Science and Technology Program of the Ministry of Housing and Urban-Rural Development of China, under grant no. 2020-S-001.

References

[1] A. Orso, G. Rothermel, Software testing: a research travelogue (2000-2014), in: Proceedings of the on Future of Software Engineering (FOSE'14), 2014, pp. 117–132.
[2] S. Anand, E.K. Burke, T.Y. Chen, J.A. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation, J. Syst. Softw. 86 (8) (2013) 1978–2001.
[3] A. Arcuri, M.Z. Iqbal, L. Briand, Random testing: theoretical results and practical implications, IEEE Trans. Softw. Eng. 38 (2) (2011) 258–277.
[4] J.W. Duran, S.C. Ntafos, An evaluation of random testing, IEEE Trans. Softw. Eng. 10 (4) (1984) 438–444.
[5] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Softw. Test. Verif. Reliab. 22 (2) (2012) 67–120.
[6] R. Huang, X. Xie, T.Y. Chen, Y. Lu, Adaptive random test case generation for combinatorial testing, in: Proceedings of the IEEE 36th Annual Computer Software and Applications Conference (COMPSAC'12), 2012, pp. 52–61.
[7] D. Cotroneo, R. Pietrantuono, S. Russo, RELAI testing: a technique to assess and improve software reliability, IEEE Trans. Softw. Eng. 42 (5) (2016) 452–475.
[8] S.H. Shin, S.K. Park, K.H. Choi, K.H. Jung, Normalized adaptive random test for integration tests, in: Proceedings of the IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW'10), 2010, pp. 335–340.
[9] J. Mayer, C. Schneckenburger, An empirical analysis and comparison of random testing techniques, in: Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06), 2006, pp. 105–114.
[10] A. Shahbazi, A.F. Tappenden, J. Miller, Centroidal Voronoi tessellations - a new approach to random testing, IEEE Trans. Softw. Eng. 39 (2) (2013) 163–183.
[11] L.J. White, E.I. Cohen, A domain strategy for computer program testing, IEEE Trans. Softw. Eng. 6 (3) (1980) 247–257.
[12] P.E. Ammann, J.C. Knight, Data diversity: an approach to software fault tolerance, IEEE Trans. Comput. 37 (4) (1988) 418–425.
[13] T.Y. Chen, F. Kuo, R.G. Merkel, T.H. Tse, Adaptive random testing: the ART of test case diversity, J. Syst. Softw. 83 (1) (2010) 60–66.
[14] T.Y. Chen, F. Kuo, D. Towey, Z. Zhou, A revisit of three studies related to random testing, Sci. China Inf. Sci. 58 (5) (2015) 1–9.
[15] T.Y. Chen, F. Kuo, H. Liu, W.E. Wong, Code coverage of adaptive random testing, IEEE Trans. Reliab. 62 (1) (2013) 226–237.
[16] T.Y. Chen, R.G. Merkel, An upper bound on software testing effectiveness, ACM Trans. Softw. Eng. Methodol. 17 (3) (2008) 16.
[17] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, X. Xia, A survey on adaptive random testing, IEEE Trans. Softw. Eng. (2019), https://doi.org/10.1109/TSE.2019.2942921.
[18] K.P. Chan, T.Y. Chen, D. Towey, Restricted random testing, in: Proceedings of the 7th European Conference on Software Quality (ECSQ'02), 2002, pp. 321–330.
[19] K.P. Chan, T.Y. Chen, D. Towey, Restricted random testing: adaptive random testing by exclusion, Int. J. Softw. Eng. Knowl. Eng. 16 (04) (2006) 553–584.
[20] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, ARTOO: adaptive random testing for object-oriented software, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), 2008, pp. 71–80.
[21] J. Lv, H. Hu, K. Cai, T.Y. Chen, Adaptive and random partition software testing, IEEE Trans. Syst. Man Cybern. 44 (12) (2014) 1649–1664.
[22] E. Nikravan, S. Parsa, Hybrid adaptive random testing, Int. J. Comput. Sci. Math. 11 (3) (2020) 209–221.
[23] K.K. Sabor, S. Thiel, Adaptive random testing by static partitioning, in: Proceedings of the 10th International Workshop on Automation of Software Test (AST'15), 2015, pp. 28–32.
[24] T.Y. Chen, F. Kuo, H. Liu, Adaptive random testing based on distribution metrics, J. Syst. Softw. 82 (9) (2009) 1419–1433.
[25] T.Y. Chen, F.-C. Kuo, R.G. Merkel, On the statistical properties of the F-measure, in: Proceedings of the 4th International Conference on Quality Software (QSIC'04), 2004, pp. 146–153.
[26] T.Y. Chen, R.G. Merkel, An upper bound on software testing effectiveness, ACM Trans. Softw. Eng. Methodol. 17 (3) (2008) 16.
[27] A. Arcuri, L.C. Briand, Adaptive random testing: an illusion of effectiveness?, in: Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11), 2011, pp. 265–275.
[28] T.Y. Chen, D.H. Huang, Z.Q. Zhou, Adaptive random testing through iterative partitioning, in: Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'06), 2006, pp. 155–166.
[29] T.Y. Chen, F. Kuo, H. Liu, W.E. Wong, Does adaptive random testing deliver a higher confidence than random testing?, in: Proceedings of the 8th International Conference on Quality Software (QSIC'08), 2008, pp. 145–154.
[30] R. Huang, C. Cui, W. Sun, D. Towey, Poster: is Euclidean distance the best distance measurement for adaptive random testing?, in: Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification, (ICST'20), 2020, pp. 406–409.
[31] C. Schneckenburger, J. Mayer, Towards the determination of typical failure patterns, in: Proceedings of the 4th International Workshop on Software Quality Assurance (SOQUA'07), 2007, pp. 90–93.
[32] T.Y. Chen, F.-C. Kuo, Z.Q. Zhou, On the relationships between the distribution of failure-causing inputs and effectiveness of adaptive random testing, in: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), 2005, pp. 306–311.

[33] F.T. Chan, T.Y. Chen, I.K. Mak, Y. Yu, Proportional sampling strategy: guidelines for software testing practitioners, Inf. Softw. Technol. 38 (12) (1996) 775–782.
[34] T.Y. Chen, H. Leung, I.K. Mak, Adaptive random testing, in: Proceedings of the 9th Asian Computing Science Conference (ASIAN'04), 2004, pp. 320–329.
[35] T.Y. Chen, F.-C. Kuo, H. Liu, On test case distributions of adaptive random testing, in: Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07), 2007, pp. 141–144.
[36] I.K. Mak, On the effectiveness of random testing, Master's thesis, University of Melbourne, Australia, 1997.
[37] T.Y. Chen, F. Kuo, R.G. Merkel, On the statistical properties of testing effectiveness measures, J. Syst. Softw. 79 (5) (2006) 591–601.
[38] R. Huang, H. Liu, X. Xie, J. Chen, Enhancing mirror adaptive random testing through dynamic partitioning, Inf. Softw. Technol. 67 (2015) 13–29.
[39] T.Y. Chen, F.-C. Kuo, Is adaptive random testing really better than random testing, in: Proceedings of the 1st International Workshop on Random Testing (RT'06), 2006, pp. 64–69.
[40] T.Y. Chen, F. Kuo, Z. Zhou, On favourable conditions for adaptive random testing, Int. J. Softw. Eng. Knowl. Eng. 17 (6) (2007) 805–825.
[41] W. Press, B.P. Flannery, S.A. Teulolsky, W.T. Vetterling, Numerical Recipes, Cambridge University Press, Cambridge, 1986.
[42] A.C.M. Collected, Algorithms from, ACM, Association for Computer Machinery, New York, 1980.
[43] J. Ferrer, F. Chicano, E. Alba, Evolutionary algorithms for the multi-objective test data generation problem, Softw. Pract. Exp. 42 (11) (2012) 1331–1362.
[44] Y.D. Liang, Introduction to Java Programming and Data Structures, Comprehensive Version, 11th ed., Pearson Education, London, UK, 2017.
[45] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678.
[46] A. Arcuri, L.C. Briand, A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Softw. Test. Verif. Reliab. 24 (3) (2014) 219–250.
[47] O. Beaumont, A. Kermarrec, E. Riviere, Peer to peer multidimensional overlays: approximating complex structures, in: Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS'07), vol. 4878, 2007, pp. 315–328.
[48] C. Mao, X. Zhan, T.H. Tse, T.Y. Chen, KDFC-ART: a KD-tree approach to enhancing fixed-size-candidate-set adaptive random testing, IEEE Trans. Reliab. 68 (4) (2019) 1444–1469.
[49] T.Y. Chen, F.-C. Kuo, R.G. Merkel, S.P. Ng, Mirror adaptive random testing, in: Proceedings of the 3rd International Conference on Quality Software (QSIC'03), 2003, pp. 4–11.
[50] K.P. Chan, T.Y. Chen, D. Towey, Adaptive random testing with filtering: an overhead reduction technique, in: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), 2005, pp. 292–299.
[51] K.P. Chan, T.Y. Chen, D. Towey, Forgetting test cases, in: Proceedings of the IEEE 30th Annual International Computer Software and Applications Conference (COMPSAC'06), 2006, pp. 485–494.
[52] T.Y. Chen, R.G. Merkel, Quasi-random testing, IEEE Trans. Reliab. 56 (3) (2007) 562–568.
[53] S. Xu, P. Xu, A quasi-best random testing, in: Proceedings of the 19th IEEE Asian Test Symposium (ATS'10), 2010, pp. 21–26.
[54] H. Liu, T.Y. Chen, Randomized quasi-random testing, IEEE Trans. Comput. 65 (6) (2016) 1896–1909.
[55] F.-C. Kuo, An indepth study of mirror adaptive random testing, in: Proceedings of the 9th International Conference on Quality Software (QSIC'09), 2009, pp. 51–58.
[56] T.Y. Chen, T.H. Tse, Y.T. Yu, Proportional sampling strategy: a compendium and some insights, J. Syst. Softw. 58 (1) (2001) 65–81.