SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2017, Vol. 60 092106:1–092106:21
doi: 10.1007/s11432-016-0087-0

Out of sight, out of mind: a distance-aware forgetting strategy for adaptive random testing

Chengying MAO[1]*, Tsong Yueh CHEN[2] & Fei-Ching KUO[2]

[1]*School of Software and Communication Engineering, Jiangxi University of Finance and Economics, Nanchang* 330013*, China;*
[2]*Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne* 3122*, Australia*

Received August 29, 2016; accepted October 31, 2016; published online April 27, 2017

Abstract Adaptive random testing (ART) achieves better failure-detection effectiveness than random testing by increasing the diversity of test cases. However, the intention of ensuring even spread of test cases inevitably causes an overhead problem. Although two basic forgetting strategies (i.e. random forgetting and consecutive retention) were proposed to reduce the computation cost of ART, they only considered the temporal distribution of test cases. In the paper, we presented a distance-aware forgetting strategy for the fixed size candidate set version of ART (DF-FSCS), in which the spatial distribution of test cases is taken into consideration. For a given candidate, the test cases out of its "sight" are ignored to reduce the distance computation cost. At the same time, the dynamic adjustment for partitioning and the second-round forgetting are adopted to ensure the linear complexity of DF-FSCS algorithm. Both simulation analysis and empirical study are employed to investigate the efficiency and effectiveness of DF-FSCS. The experimental results show that DF-FSCS significantly outperforms the classical ART algorithm FSCS-ART in efficiency, and has comparable failure-detection effectiveness. Compared with two basic forgetting methods, DF-FSCS is better in both efficiency and effectiveness. In contrast with a typical linear-time ART algorithm RBCVT-Fast, our algorithm requires less computational overhead and exhibits the similar failure-detection capability. In addition, DF-FSCS has more reliable performance than RBCVT-Fast in detecting failures for the programs with high-dimensional input domain.

Keywords adaptive random testing, software testing, test cases, computational overhead, diversity

Citation Mao C Y, Chen T Y, Kuo F-C. Out of sight, out of mind: a distance-aware forgetting strategy for adaptive random testing. Sci China Inf Sci, 2017, 60(9): 092106, doi: 10.1007/s11432-016-0087-0

1 Introduction

Compared with white-box testing, black-box testing shows a great advantage in terms of the test case selection cost. Especially for random testing (RT) [1], it only needs the information about program input domain. However, RT has been commonly criticized for its low failure-detection capability. In order to overcome this problem, *adaptive random testing* (ART) [2, 3] has been proposed by Chen et al. to enhance the failure-detection effectiveness of random testing [4].

* Corresponding author (email: maochy@yeah.net)

© Science China Press and Springer-Verlag Berlin Heidelberg 2017

Some previous experimental studies have shown that, in order to achieve a higher failure-detection probability, test cases need to be evenly spread across the input domain, that is to achieve the diversity of test cases [3, 5–7]. As an improved version of random testing, ART considers how to make better diversity of test cases without loosing randomness [8–10]. The increase of the diversity, however, often implies an increase of cost on selecting test cases. As a consequence, some ART methods have a potentially high computation overhead [11]. For example, *fixed size candidate set* version of the ART (abbreviated as FSCS-ART) [2] has been validated as an effective enhancement to RT. Unfortunately, it has heavy overhead in computing the distance from each candidate to already executed test cases. Although two forgetting strategies [12], including random forgetting and consecutive retention for the executed test cases, have been proposed to reduce the overhead of ART, they only consider random selection or the temporal order of test cases, and ignore an important feature of the diversity of test cases—spatial distribution—in distance computation.

In fact, the spatial distribution of executed test cases is an important information for constructing effective "forgetting" strategies. Given a candidate in FSCS-ART, to reduce the overhead of distance computation, test cases beyond its "sight" could be forgotten for a cost-effective computation as they are unlikely to be its nearest neighbours. In this paper, we will enhance the efficiency of FSCS-ART by applying the grid partitioning, indexing and neighbours querying [13] to exclude unnecessary distance computation. In our approach, only the test cases lying in the neighbour region (cells) of a candidate, rather than *all* already executed test cases, should be kept in the memory for distance computation. In order to keep a constant cost for distance computation in the whole process, dynamic adjustment for partitioning is introduced to control the number of test cases within each neighbour region. Although the test cases beyond the "sight" of a candidate are forgotten, the failure-detection capability of our distance-aware forgetting FSCS-ART algorithm (referred to as DF-FSCS) is still comparable to the FSCS-ART. More importantly, the overhead of DF-FSCS is significantly lower than that of FSCS-ART. Compared with another linear-time ART algorithm RBCVT-Fast [14], it also has significant advantage in computational overhead. In addition, DF-FSCS exhibits a better failure-detection capability than two basic forgetting strategies.

2 Preliminaries

2.1 Program failure patterns

As reported in the previous studies [15–17], test inputs which can reveal programs failures (i.e., *failure-causing inputs*) usually tend to cluster together in one or more contiguous regions (referred to as *failure regions*). The patterns of these failure regions can be roughly classified into three types [18]: block pattern, strip pattern and point pattern. As illustrated in Figure 1, in *block pattern*, the failure-causing inputs form one or more closed regions. From the viewpoint of program structure, failures in this pattern are often triggered by the computational faults which occur when the expression in an assignment statement is incorrect. *Strip pattern* is common for domain errors [17], the corresponding failures may be attributed to predicate fault in a branch. In *point pattern*, the feature of such errors is that the failure-causing inputs appear to scatter over a large part of the input domain.

According to the notation used by Chen et al. [2], for a program and its input domain, the rate of failure-causing inputs over the whole input domain is defined as *failure rate* (θ). In that way, the probability of detecting at least one failure in a program can be depicted by this statistic. Obviously, the lower failure rate means the more difficult to detect faults in the program.

2.2 FSCS-ART and its overhead problem

ART [3] can effectively increase the chance of failure detection through an even distribution of test cases across the input domain. As the first and most extensively studied ART algorithms, FSCS-ART [2] has exhibited good effectiveness of failure detection.

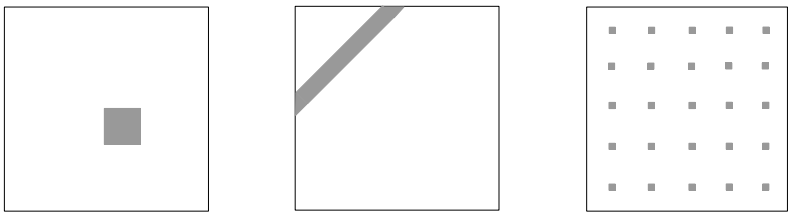

Figure 1 The typical failure patterns in a two-dimensional input domain. Here, the shaded areas represent failure-causing inputs. (a) Block pattern; (b) strip pattern; (c) point pattern.

In FSCS-ART, two sets of test cases, namely, the executed test cases (TS) and the candidate set (C) are adopted for distance computation. The basic principle is to find the best candidate from C, which is the farthest from all elements of TS, as the next test case. Suppose $\mathrm{TS} = \{\mathrm{tc}_1, \mathrm{tc}_2, \ldots, \mathrm{tc}_n\}$ and $C = \{c_1, c_2, \ldots, c_s\}$, a criterion of FSCS-ART is to choose the element c_h such that for all $k \in \{1, 2, \ldots, s\}$,

$$\min_{j=1}^{n} \mathrm{dist}(c_h, \mathrm{tc}_j) \geqslant \min_{j=1}^{n} \mathrm{dist}(c_k, \mathrm{tc}_j), \tag{1}$$

where dist refers to the Euclidean distance for numeric inputs.

The process of FSCS-ART can be described as follows: initially, the first test case is randomly selected from input domain. Subsequently, to select an additional test case, it randomly generates a candidate set C according to the uniform distribution. Then, the candidate that is the most "distant" from previously executed test inputs is selected as the next test case. The above process is repeated until the termination criterion is met. Such criterion can be expressed as: a failure is revealed by test case, or the predefined size of test suite is reached.

Although FSCS-ART has a good fault-revealing capability, the high computation cost brings some adverse effects on its applications. To generate jth test case ($1 < j \leqslant n$), for each candidate, the algorithm has to compute the distances to all previously executed $(j-1)$ test cases, so its complexity is $O(\text{FSCS-ART}) = O(\sum_{j=2}^{n}(j-1)\cdot s) = O(s \cdot n^2)$, where s is the size of candidate set, and n is the size of test suite.

In the past, two forgetting strategies [12] were proposed to reduce the number of executed test cases for consideration in distance computation. The first forgetting strategy (*Random Forgetting*) is to randomly select a subset of TS, namely *memory set* MS, where $\mathrm{MS} = \{\mathrm{tc}'_1, \mathrm{tc}'_2, \ldots, \mathrm{tc}'_m\}$, m is called *memory size* and $m < n$. In this paper, we denote this strategy for FSCS-ART as RF-FSCS. The second strategy (*Consecutive Retention*), called CR-FSCS here, is to keep the most recently selected m test cases in the executed sequence in FSCS-ART as memory set. It is not hard to find that, the complexities for both RF-FSCS and CR-FSCS are of the order $O(s \cdot m \cdot n)$.

In selecting a part of already executed test cases for distance computation, the above two forgetting strategies do not consider the *spatial* locations of candidates and test cases. Since the "unforgotten" test cases may not well represent all previously executed test cases, the selected candidate may not be an ideal test case. In this paper, we present a distance-aware forgetting strategy for FSCS-ART so as to overcome the above weakness.

3 Distance-aware forgetting strategy

3.1 Basic concepts

The fundamental cause of high overhead of FSCS-ART lies in that each candidate has to calculate the distances to all previously executed test cases. When the executed test cases reach a certain number, the overhead of distance computation will become a heavy burden to select the next test case. Intuitively, in order to reduce the computational overhead, some test cases should be "forgotten" for distance compu-

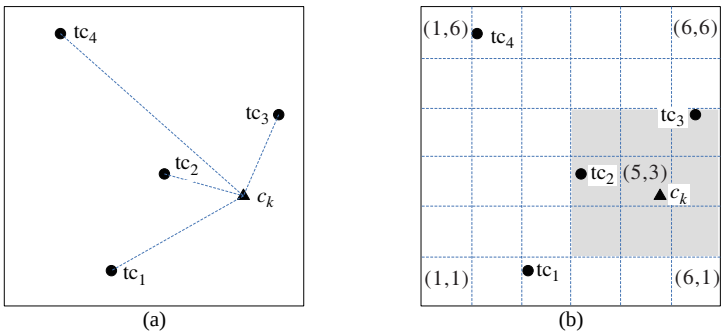

Figure 2 An illustration of distance-aware forgetting strategy for ART. (a) The basic FSCS-ART; (b) distance computation within the sight of candidate point c_k.

tation. Here, inspired by the concept of localization [19], we attempt to construct an improved version of FSCS-ART by applying "forgetting" strategy and considering the spatial distribution of test cases.

To illustrate our approach, consider the example shown in Figure 2(a). Suppose four test cases (from tc_1 to tc_4) have been executed and c_k is a candidate. In the original FSCS-ART algorithm, the distances from c_k to all four test cases should be computed one by one in order to find which $\mathrm{dist}(c_k, \mathrm{tc}_j)$, where $1 \leqslant j \leqslant 4$, is the shortest. Obviously, the overhead will become more and more expensive with the growth of the number of executed test cases. In order to overcome this problem, we can store the test cases lying in the neighbour cells of a given candidate as memory set, and then discard the others in TS so as to reduce the overhead of distance computation. Here, grid partitioning is adopted to determine the neighbour region of the given candidate. In this example, the whole input domain is partitioned into 6×6 grids (refer to Figure 2(b)), which are encoded from the bottom left corner (i.e. (1, 1)) to the top right corner (i.e. (6, 6)). Based on this partition, it is easy to locate the candidate c_k into grid (5, 3). Accordingly, the neighbour cells of c_k can be represented by the shadow area, that is, the nine cells from (4, 2) to (6, 4). Considering the existing four test cases, only two of them (i.e. tc_2 and tc_3) are located in c_k's neighbour region, and the others are beyond its scope. In this way, we can significantly reduce the computational overhead, but still keep acceptable accuracy on the shortest distance from c_k to the test suite.

Without loss of generality, suppose the length in each dimension is equal. For the case of input domain with unequal dimensions, we will discuss it in Subsection 6.1.

Definition 1 (cells in input domain)**.** For a d-dimensional input domain, if we divide each dimension into p equal-length parts, the whole domain can be partitioned into p^d cells, such as square for 2-dimensional space and cube for 3-dimensional space. Each cell can be indexed by $(\mathrm{sn}_1, \mathrm{sn}_2, \ldots, \mathrm{sn}_d)$, where sn_i is the position identifier of partitioning in the ith $(1 \leqslant i \leqslant d)$ dimension, obviously $1 \leqslant \mathrm{sn}_i \leqslant p$.

Once a candidate has been located into a cell, its neighbour region can be defined as follows.

Definition 2 (neighbour region)**.** In a d-dimensional input domain, assume candidate c_k lies in a cell named $\mathrm{cell}(c_k) = (\mathrm{sn}_{k1}, \mathrm{sn}_{k2}, \ldots, \mathrm{sn}_{kd})$. Its neighbour region $\mathrm{neig}(c_k)$ can be defined as below.

$$\mathrm{neig}(c_k) = \{(\mathrm{sn}'_{k1}, \mathrm{sn}'_{k2}, \ldots, \mathrm{sn}'_{kd}) \mid \mathrm{sn}'_{ki} \in \{\mathrm{sn}_{ki} - 1, \mathrm{sn}_{ki}, \mathrm{sn}_{ki} + 1\}, 1 \leqslant \mathrm{sn}'_{ki} \leqslant p, \text{and } 1 \leqslant i \leqslant d\}. \quad (2)$$

According to above definition, it is easy to find a property about the number of cells within the neighbour region (denoted as $|\mathrm{neig}(c_k)|$): $|\mathrm{neig}(c_k)| \leqslant 3^d$.

Definition 3 (test cases within the neighbour region)**.** Given a candidate c_k, its neighbour region can be easily identified by (2), i.e. $\mathrm{neig}(c_k)$. Suppose the current test suite is TS, then the set of test cases lying in $\mathrm{neig}(c_k)$ is denoted as $\mathrm{TS}_{\mathrm{neig}(c_k)}$. Obviously, $\mathrm{TS}_{\mathrm{neig}(c_k)} \subseteq \mathrm{TS}$.

Although the FSCS-ART algorithm uses the spatial positions of candidates and test cases for distance computation, but this location information has not been fully utilized yet. That is to say, the relative position relation between a given candidate and the executed test cases has not been used to ignore some distance computation. Different from the FCST-ART, our algorithm only takes the test cases in

$\mathrm{TS}_{\mathrm{neig}(c_k)}$ into consideration for distance computation, so the corresponding overhead can be reduced to a lower level.

3.2 Algorithm of DF-FSCS

3.2.1 *Localization and indexing of test cases*

In a d-dimensional input space, for any test case $\mathrm{tc}_j \in \mathrm{TS}$, coordinates of $\mathrm{tc}_j = (x_{j1}, x_{j2}, \ldots, x_{jd})$ can be used to position tc_j in a specific cell. The ith coordinate of tc_j, i.e. x_{ji}, can be converted to the position of the corresponding cell in ith dimension.

$$\mathrm{sn}_{ji} = \left\lceil \frac{x_{ji} - \mathrm{lb}_i}{l_{\mathrm{side}}} \right\rceil, \tag{3}$$

where $\lceil\ \rceil$ is the ceiling operation, l_{side} is the side length of basic cell, and lb_i is the lower boundary of input domain in the ith dimension.

By (3), any test case tc_j can be easily represented by a cell identifier $\mathrm{sn}_j = (\mathrm{sn}_{j1}, \mathrm{sn}_{j2}, \ldots, \mathrm{sn}_{jd})$. Since a cell can contain several test cases, different test cases may have the same cell identifier.

In order to quickly find the neighbours for a given candidate, we have to convert all test cases into the identifers formatted by cell position. Meanwhile, a convenient indexing structure should also be designed for finding test cases in a specific cell. Here, we adopt the array of lists to store all test cases in TS. This structure consists of the following two main parts: the head is an array of cell identifiers varying from $(1, 1, \ldots, 1)$ to $(p_1, p_2, \ldots, p_d)$, where p_i is the partitioning number in the ith dimension ($1 \leqslant i \leqslant d$). For each cell identifier in head, it has a follow-up list to store the test cases in the corresponding cell.

3.2.2 *Dynamic adjustment for partitioning*

Based on grid partitioning, our algorithm can ignore the test cases beyond the "sight" of a candidate for distance computation. However, if the partitioning remains unchanged, the test cases in a cell will continuously increase along with the selection process. As a consequence, the computational overhead for each candidate cannot be kept as a constant.

To overcome this problem, we have to dynamically adjust the partitioning with the increase of test suite size. The adjustment strategy is defined as follows: at the initial stage, each dimension of input domain is divided into p_0 (e.g. 1, 3 or 6) equal intervals, and φ_0 ($= p_0^d$) basic cells can be yielded. Here, we call the initial partitioning as the 0th partition. During test case selection process, once the number of executed test cases reaches a threshold, the partitioning should be adjusted: each cell in the last-round partitioning is further divided into 2^d small equal cells. Then, the number of cells after the tth partition will satisfy the following relation: $\varphi_t = 2^d \cdot \varphi_{t-1} = 2^{td} \cdot \varphi_0$.

Once the dynamic partitioning is applied, all test cases in the current test suite should be reassigned into the new cells. For each reassignment, the time complexity is equal to the size of current test suite. Thus, in order to reduce additional cost caused by the dynamic adjustment, the partitioning times should be controlled in a reasonable range.

In our method, the condition of dynamic adjustment is set as follows: once the size of test suite reaches τ times of the number of cells, a new-round partitioning should be performed. Here, τ is a pre-set constant (e.g. $\tau = 5$). That is to say, the tth partitioning shall be done if the test suite size reaches the threshold of $\tau \cdot 2^{(t-1)d} \cdot \varphi_0$. It is easy to see that the serial number (n) of test case is a staircase function of the iteration number (T) of partitioning. Specifically, suppose the nth test case is selected and executed between the Tth and $(T+1)$th partitioning, then n and T should have the relation as follow: if $n < \tau \cdot \varphi_0$, $T = 0$. Otherwise,

$$T = \left\lfloor \frac{1}{d} \log_2 \left(\frac{n}{\tau \cdot \varphi_0} \right) \right\rfloor + 1, \tag{4}$$

where $\lfloor\ \rfloor$ is the floor operation, d is the dimension number of input domain, φ_0 is the number of cells in the initial partitioning.

Despite that the above strategy can reduce the cost for reassigning the previous test cases, it may lead to uncertainty in the size of test cases lying in the neighbour region of candidate c_k, which is denoted as $|\mathrm{TS}_{\mathrm{neig}(c_k)}|$ in the previous part. Accordingly, the overhead of computing distances from $\mathrm{TS}_{\mathrm{neig}(c_k)}$ to c_k will not have a fixed upper-bound. In order to ensure a constant cost, the number of test cases for distance computation must be controlled by the second-round of forgetting. From the whole scope of input domain, the number of test cases is controlled under the product of τ and number of cells produced by dynamic partitioning. In ART, the main task is to realize the uniform distribution of test cases across input domain, so the restriction of test case number in a partial region should be consistent with this intuition. Since the number of cells in a candidate's neighbourhood is 3^d, the corresponding upper bound of the number of test cases within such region can be set as $\tau \cdot 3^d$.

(1) If $|\mathrm{TS}_{\mathrm{neig}(c_k)}| \leqslant \tau \cdot 3^d$, all test cases in $\mathrm{TS}_{\mathrm{neig}(c_k)}$ are used for distance computation with respect to c_k.

(2) Otherwise, select $\tau \cdot 3^d$ test cases from $\mathrm{TS}_{\mathrm{neig}(c_k)}$ according to a random strategy.

As mentioned above, τ is a pre-set constant before the dynamic partitioning. An in-depth analysis shows that τ has a impact on partition times. If it is set too low, it will cause more partitioning and re-location of test cases. On the contrary, it will lead to an increase of distance computation overhead for each candidate. For this reason, we choose a moderate value (which is $\tau = 5$ in our experiments) for it. According to this setting, take a 2-dimensional input domain for example, the upper bound of the number of test cases used for distance computation for a candidate is $5 \times 3 \times 3 = 45$.

In order to facilitate the expression, we define the selected subset from $\mathrm{TS}_{\mathrm{neig}(c_k)}$ as below.

Definition 4 (the controlled test subset within neighbour region)**.** During the distance computation for candidate c_k, we control the test cases sampled from $\mathrm{TS}_{\mathrm{neig}(c_k)}$ not exceeding $\tau \cdot 3^d$. Here, the selected test cases are denoted collectively as the controlled test subset within neighbour region, i.e. $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$. Obviously, $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)} \subseteq \mathrm{TS}_{\mathrm{neig}(c_k)}$.

Thus, the complexity of computing the approximately shortest distance from candidate c_k to $\mathrm{TS}_{\mathrm{neig}(c_k)}$ will be kept by the order of $|\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}|$. Since $|\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}| \leqslant \tau \cdot 3^d$, the time complexity of distance computation is a constant cost.

3.2.3 *Algorithm description*

Based on the above definitions and analysis, a distance-aware forgetting strategy for FSCS-ART can be defined, which is referred to as DF-FSCS algorithm.

At the initial stage (lines 1–4), the first test case is randomly selected from the whole input domain. Meanwhile, each dimension of input domain is divided into p_0 equal parts, then the number of cells in the initial partitioning can be calculated as $\varphi_0 = p_0^d$. Consequently, once a test case is selected, it will be added into TS and will be indexed and assigned to the relevant cell (lines 6 and 7). While the size of test cases is increasing, it needs to check whether the basic cell should be further partitioned so as to control the number of test cases in a cell (lines 8–12). From lines 13 to 27, it is to randomly generate s candidates and select an appropriate one as the next test case. For each candidate (lines 14–26), locate the cell in which it lies (i.e. $\mathrm{cell}(c_k)$), and find its neighbour region $\mathrm{neig}(c_k)$ using the indexing structure. If there is no test case in $\mathrm{neig}(c_k)$, c_k is treated as a test case directly (lines 17–20). Otherwise, line 21 calls procedure `SelectNeighbours(`c_k, $\mathrm{TS}_{\mathrm{neig}(c_k)}$`)` to select a subset ($\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$) of test cases from $\mathrm{neig}(c_k)$, whose size is less than or equal to $\tau \cdot 3^d$. Subsequently, the distances from test cases in $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$ to c_k are computed one by one, and to determine the shortest one. On line 27, the candidate whose approximate shortest distance is the largest is found and viewed as the next test case.

In procedure `SelectNeighbours(`c_k, $\mathrm{TS}_{\mathrm{neig}(c_k)}$`)`, the test cases in $\mathrm{cell}(c_k)$ are preferentially selected as neighbours of c_k. If the number of test cases in $\mathrm{cell}(c_k)$ is greater than threshold $\tau \cdot 3^d$, randomly select $\tau \cdot 3^d$ ones from them to form subset $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$ (lines 3 and 4). Otherwise (lines 6 and 7), add all test cases in $\mathrm{cell}(c_k)$ into $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$, and then randomly select some test cases within c_k's neighbour cells to replenish the subset. Of course, the size of $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$ is restricted to $\tau \cdot 3^d$.

Algorithm DF-FSCS

Input: (1) The size of candidate set, denoted as s ($s > 1$).
(2) The dimension number (d), and the initial partition number (p_0) for each dimension.
(3) The pre-set threshold (τ) of the average number of test cases in a cell.

Output: The set of test cases TS.

1: Input parameters s, d, p_0 and τ;
2: Set $p = p_0$, $n = 0$, $t = 0$ and TS = {};
3: Partition each dimension into p_0 equal parts, $\varphi_0 = p_0^d$, and construct a corresponding indexing structure for cells;
4: Randomly select a test case tc from the input domain;
5: **while** (termination condition does not satisfy) **do**
6: Add tc into TS, and increment n by 1;
7: Localize the cell of tc, and add tc into the indexing structure;
//dynamic adjustment for partitioning from line 8 to 12
8: **if** $n == \tau \cdot 2^{td} \cdot \varphi_0$ **then**
9: $p = 2p$, $t = t + 1$;
10: Re-partition each dimension into p equal parts, and construct a new indexing structure for it;
11: Re-locate all test cases in TS into the new indexing structure;
12: **end if**
13: Randomly generate s candidates $c_1, c_2, \ldots, c_s$ from the input domain;
14: **for** each candidate c_k, where $k = 1, 2, \ldots, s$ **do**
15: Localize the cell of c_k, and find its neighbour region neig(c_k) according to (2);
16: Collect all test cases lying in neig(c_k) to from set $\mathrm{TS}_{\mathrm{neig}(c_k)}$;
17: **if** $\mathrm{TS}_{\mathrm{neig}(c_k)} = \emptyset$ **then**
18: tc = c_k;
19: **break**; //exit the current loop from line 14 to 26
20: **end if**
21: Call **procedure** `SelectNeighbours`(c_k,$\mathrm{TS}_{\mathrm{neig}(c_k)}$) to get the controlled test subset from $\mathrm{TS}_{\mathrm{neig}(c_k)}$, denote it as $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$;
22: **for** each test case tc_j in $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$ **do**
23: Calculate the distance from c_k to tc_j, and denote it as d_{kj};
24: **end for**
25: Find the shortest one from $\{d_{kj}\}$ ($1 \leqslant j \leqslant |\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}|$), and denote it as $\mathrm{dist}_{\mathrm{ap}}(c_k, \mathrm{TS})$;
26: **end for**
27: Find the candidate whose $\mathrm{dist}_{\mathrm{ap}}(c_k, \mathrm{TS})$ ($1 \leqslant k \leqslant s$) is the largest one, and set it as tc;
28: **end while**
29: **return** TS;

Procedure `SelectNeighbours`(c_k,$\mathrm{TS}_{\mathrm{neig}(c_k)}$)

1: $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)} = \{\}$;
2: Count test cases lying in cell(c_k), denote it as $\mathrm{TS}_{\mathrm{cell}(c_k)}$; //cell($c_k$) is the cell in which c_k lies.
3: **if** $|\mathrm{TS}_{\mathrm{cell}(c_k)}| \geqslant \tau \cdot 3^d$ **then**
4: Randomly select $\tau \cdot 3^d$ test cases from $\mathrm{TS}_{\mathrm{cell}(c_k)}$, and add them in $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$;
5: **else**
6: $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)} = \mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)} \bigcup \mathrm{TS}_{\mathrm{cell}(c_k)}$;
7: Randomly select $(\tau \cdot 3^d - |\mathrm{TS}_{\mathrm{cell}(c_k)}|)$ test cases from neighbour region neig(c_k) except cell(c_k), and add them in $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$;
8: **end if**
9: **return** $\mathrm{TS}^{\mathrm{up}}_{\mathrm{neig}(c_k)}$;

In our DF-FSCS algorithm, the neighbour region in form of grid cells can easily determine which test cases locate in the candidate's neighbourhood and thus can greatly reduce the distance computation overhead. Nevertheless, the nearest already executed test cases of a candidate may locate outside these grid cells. Thus, DF-FSCS may not identify the genuine max-min candidate as the next test case but FSCS-ART always does.

4 Complexity analysis for DF-FSCS

4.1 The worst-case computational cost

The overhead of DF-FSCS is mainly due to the following two aspects: cost for dynamic partitioning ($C_{\rm dp}$), and cost for computing distance from candidates to existing test cases ($C_{\rm dc}$).

In algorithm DF-FSCS, although the while loop (lines 5–28) could be executed n times, the times of dynamic partitioning and adjustment are much less due to the repartition condition on line 8. As analyzed in Subsection 3.2.2, the times of partitioning adjustment (lines 8–12) are T as shown in (4). Consequently, the dominant task is to reassign all existing test cases into the new indexing structure. According to the partitioning condition, the cost of the tth reassignment for test cases is $\tau \cdot 2^{(t-1)d} \cdot \varphi_0$, where $1 \leqslant t \leqslant T$. Thus, the complexity for dynamic partitioning and adjustment ($C_{\rm dp}$) can be calculated as

$$C_{\rm dp} = \sum_{t=1}^{T} 2^{(t-1)d} \cdot \tau \cdot \varphi_0 = \tau \cdot \varphi_0 \cdot \frac{2^{Td}-1}{2^d-1}. \tag{5}$$

Refer to (4), $T = \lfloor \frac{1}{d} \log_2 (\frac{n}{\tau \cdot \varphi_0}) \rfloor + 1 \leqslant \frac{1}{d} \log_2 (\frac{n}{\tau \cdot \varphi_0}) + 1$, thus,

$$C_{\rm dp} \leqslant \frac{1}{2^d-1} \left(2^d \cdot n - \tau \cdot \varphi_0\right). \tag{6}$$

Here, d, φ_0 and τ are all the given constants. $C_{\rm dp}$ is of the order of $O(n)$, that is, the linear time complexity with respect to the size of test suite.

On the other hand, the complexity of distance computation ($C_{\rm dc}$) can be analyzed as below. For each candidate c_k ($1 \leqslant k \leqslant s$), the cost for locating its corresponding cell and identifying $\mathrm{TS}_{\mathrm{neig}(c_k)}$ (the test cases within its neighbour region) is of the order of $O(3^d)$ according to our indexing structure. For procedure `SelectNeighbours()`, the cost for selecting the controlled test subset from $\mathrm{TS}_{\mathrm{neig}(c_k)}$ is of the order of $O(\tau \cdot 3^d)$. And, the distance computation cost for each candidate is also $O(\tau \cdot 3^d)$. Therefore, to generate an n-size test suite, $C_{\rm dc}$ is of the order of $O(\tau \cdot 3^d \cdot s \cdot n)$.

It is not hard to find that the cost of dynamic partitioning and adjustment (i.e., $C_{\rm dp}$) is much less than that of distance computation (i.e., $C_{\rm dc}$). Therefore, the time complexity of DF-FSCS is mainly determined by the latter, that is, $C_{\text{DF-FSCS}}$ is of the order of $O(\tau \cdot 3^d \cdot s \cdot n)$. Therefore, the complexity of DF-FSCS is also of the order of $O(n)$.

4.2 The average computational cost

For each candidate, $\tau \cdot 3^d$ is just an upper-bound of distance computation overhead from it to the test cases within its neighbour region. In fact, the real computation overhead varies from 0 to $\tau \cdot 3^d$. Hence, the above analyzed complexity is only for the worst case scenario. Now, we investigate the average complexity of DF-FSCS.

When the algorithm tries to select the jth test case, the number of the already executed test cases should be $(j-1)$. At this moment, the whole input domain should have been partitioned $\lfloor \frac{1}{d} \log_2 (\frac{j-1}{\tau \cdot \varphi_0}) \rfloor + 1$ times. Accordingly, the number of cells should be $\varphi_0 \cdot 2^{(\lfloor \frac{1}{d} \log_2 (\frac{j-1}{\tau \cdot \varphi_0}) \rfloor + 1)d}$. On average, the number of test cases lying in a cell can be calculated as $(j-1)/(\varphi_0 \cdot 2^{(\lfloor \frac{1}{d} \log_2 (\frac{j-1}{\tau \cdot \varphi_0}) \rfloor + 1)d})$. Then, for each candidate c_k ($1 \leqslant k \leqslant s$), since $|\mathrm{neig}(c_k)| \leqslant 3^d$, the average number of test cases lying in its neighbour region can be expressed as $|\overline{\mathrm{TS}}_{\mathrm{neig}(c_k)}|$ here.

(1) If $j \leqslant \tau \cdot \varphi_0$, $|\overline{\mathrm{TS}}_{\mathrm{neig}(c_k)}| \leqslant \frac{(j-1)3^d}{\varphi_0}$.

(2) Otherwise,

$$\left|\overline{\mathrm{TS}}_{\mathrm{neig}(c_k)}\right| \leqslant \frac{(j-1)3^d}{\varphi_0 \cdot 2^{(\lfloor \frac{1}{d} \log_2 (\frac{j-1}{\tau \cdot \varphi_0}) \rfloor + 1)d}}. \tag{7}$$

Thus, on average, there are $|\overline{\mathrm{TS}}_{\mathrm{neig}(c_k)}|$ test cases should be taken into consideration for each candidate in the distance computation. Accordingly, the average overhead of distance computation (denoted as

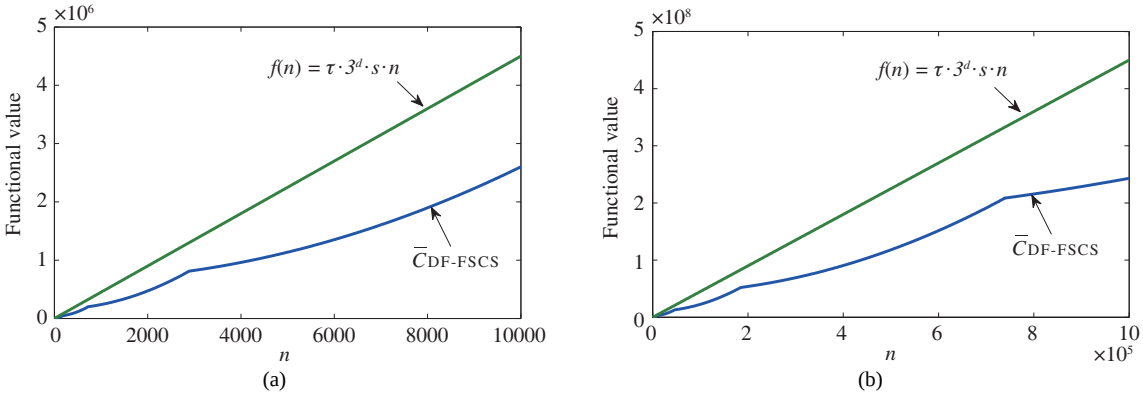

Figure 3 (Color online) The trend of the average computation overhead $\overline{C}_{\text{DF-FSCS}}$. Here, the parameters are set as follows: $\varphi_0 = 9$, $\tau = 5$, $d = 2$, $s = 10$. (a) n from 0 to 10^4; (b) n from 0 to 10^6.

$\overline{C}_{\text{dc}}$) in DF-FSCS can be measured as below.

$$\overline{C}_{\text{dc}} = \sum_{j=1}^{n} |\overline{\text{TS}}_{\text{neig}(c_k)}| \cdot s = \sum_{j=1}^{\tau\cdot\varphi_0} \frac{(j-1)3^d \cdot s}{\varphi_0} + \sum_{j=(\tau\cdot\varphi_0+1)}^{n} \frac{(j-1)3^d \cdot s}{\varphi_0 \cdot 2^{(\lfloor \frac{1}{d}\log_2(\frac{j-1}{\tau\cdot\varphi_0})\rfloor+1)d}}. \tag{8}$$

Based on the above analysis, the average time complexity of DF-FSCS consists of the cost of dynamic partitioning (C_{dp} shown in (5)) and the average cost of distance computation ($\overline{C}_{\text{dc}}$ shown in (8)), that is,

$$\begin{aligned} O(\overline{C}_{\text{DF-FSCS}}) &= C_{\text{dp}} + \overline{C}_{\text{dc}} \\ &= \tau \cdot \varphi_0 \cdot \frac{2^{(\lfloor \frac{1}{d}\log_2(\frac{n}{\tau\cdot\varphi_0})\rfloor+1)d} - 1}{2^d - 1} + \sum_{j=1}^{\tau\cdot\varphi_0} \frac{(j-1)3^d \cdot s}{\varphi_0} \\ &\quad + \sum_{j=(\tau\cdot\varphi_0+1)}^{n} \frac{(j-1)3^d \cdot s}{\varphi_0 \cdot 2^{(\lfloor \frac{1}{d}\log_2(\frac{j-1}{\tau\cdot\varphi_0})\rfloor+1)d}}. \end{aligned} \tag{9}$$

In order to understand the behaviour of $\overline{C}_{\text{DF-FSCS}}$ with respect to n, we draw its curves in different intervals of n value. According to the results shown in Figure 3, $\overline{C}_{\text{DF-FSCS}}$ has a wave-like increase with growth of n value. The turning point (also known as singular point) usually appears when n is exactly divided by $\tau \cdot \varphi_0 \cdot 2^{(t-1)d}$, where t $(= 1, 2, \dots)$ is the serial number of partitioning.

Since $\lfloor \frac{1}{d}\log_2(\frac{j-1}{\tau\cdot\varphi_0})\rfloor + 1 > \frac{1}{d}\log_2(\frac{j-1}{\tau\cdot\varphi_0})$, we have $2^{(\lfloor \frac{1}{d}\log_2(\frac{j-1}{\tau\cdot\varphi_0})\rfloor+1)d} > \frac{j-1}{\tau\cdot\varphi_0}$. As a result, we can further prove that

$$O(\overline{C}_{\text{dc}}) < O(\tau \cdot 3^d \cdot s \cdot n). \tag{10}$$

Further, we know that C_{dp} is of the order of $O(n)$ according to the (6). Thus, we can comprehensively draw the upper bound of the average computation overhead of DF-FSCS as below.

$$O(\overline{C}_{\text{DF-FSCS}}) < O(\tau \cdot 3^d \cdot s \cdot n). \tag{11}$$

In order to verify the above theoretical analysis, the line of the worst-case computational cost ($C_{\text{DF-FSCS}}$), i.e. $f(n) = \tau \cdot 3^d \cdot s \cdot n$, is also drawn in Figure 3. As seen, the curve of $\overline{C}_{\text{DF-FSCS}}$ always lies under the line of $C_{\text{DF-FSCS}}$.

5 Simulation analysis

5.1 Experimental setup and measures

In DF-FSCS algorithm, a new kind of "forgetting" strategy is designed to reduce the computational overhead of FSCS-ART algorithm by considering the spatial distribution of executed test cases. This

algorithm should be compared with an original version of FSCS-ART algorithm, as well as two other basic forgetting algorithms RF-FSCS and CR-FSCS where "execution time" is taken as the forgetting criterion rather than "spatial distribution" of test cases. DF-FSCS should also be compared with RBCVT-Fast [14], an ART algorithm having linear-order computation overhead.

In the experiment, 2-dimensional space (i.e., $d = 2$), as a typical and straightforward representative, was treated as input domain. The intervals of X-coordinate and Y-coordinate were both set as $[-5000, 5000]$. The size of candidate set was set as $s = 10$. In DF-FSCS algorithm, the parameters were setup as below: the input domain was initially partitioned into $\varphi_0 = 3 \times 3 = 9$ cells, and the threshold (τ) of test cases in a cell was assigned to 5. Accordingly, the memory size in basic forgetting algorithms should be $m = \tau \cdot 3^d = 45$. For RBCVT-Fast algorithm, its implementation was directly adopted from source code provided by Shahbazi et al.[1)], and the parameters were assigned with the recommended settings in [14]. While considering the efficiency of each algorithm, the repeated trials in experiment were set as 1000. To compare the failure-detection effectiveness, the repeated times were set as 10000 for each algorithm. The experimental environment was Windows 7 with 3.1 GHz and 8 GB memory. All algorithms were implemented in Java and run on the Eclipse platform with JDK 1.7.

For the sake of comparison, the following two metrics were adopted to measure the effectiveness of algorithms [20]: (1) F-measure: the expected number of test cases to detect the first failure. This measure is usually suitable to the algorithms of incrementally generating test cases followed by immediately executing them. In order to measure the improvement of ART algorithm with respect to the general random testing, a variant form of F-measure is usually used to reflect the improvement: F-ratio $= \frac{F_{\text{art}}}{F_{\text{rt}}}$, where F_{art} and F_{rt} represent the F-measure values of ART and general RT, respectively. (2) P-measure: the probability of detecting at least one failure. This measure is suitable for the testing methods used to generate a fixed-size test suite. Here, we adopted it to compare the effectiveness between DF-FSCS and RBCVT-Fast.

According to the settings of X-coordinate and Y-coordinate, a $10^4 \times 10^4$ square is used as input domain in the simulation analysis phase. Once the failure rate θ and failure pattern are predefined, one or several small regions (i.e. failure regions) whose total area equals to $\theta \times 10^8$ is randomly placed in the input domain. A failure is said to be found if a point inside the failure regions is selected as a test case.

Since the main concern of our DF-FSCS algorithm is to reduce the computational overhead of the typical FSCS-ART, it needs to be compared with FSCS-ART as well as other three ART methods in linear time complexity. The first research question (i.e. RQ1) was designed to find out how much overhead reduction DF-FSCS algorithm can deliver. Besides, three other questions RQ2–RQ4 were designed to find out whether failure-detection capability of FSCS-ART is compromised after the overhead-reduction.

5.2 Analysis on computation efficiency

The main purpose of this work is to reduce the overhead of FSCS-ART algorithm, so computation efficiency is the premier aspect with which we should be concerned.

RQ1: Is DF-FSCS much faster than FSCS-ART? And, is it also faster than other typical linear-time algorithms (e.g., RF-FSCS, CR-FSCS and RBCVT-Fast)?

In the experiment, we varied n from 100 to 20000, the computation times of five algorithms were collected in Table 1. Obviously, the time of DF-FSCS is gradually increasing in the way of approximately linear with the growth of n. When n reaches to 20000, the computation time of DF-FSCS is just only 0.1549 s. However, the time of FSCS-ART follows a quadratic relation with test suite size. Its computation time will reach up to 68.62 s when n is 20000. As shown in Figure 4(a), RBCVT-Fast's time is greater than that of FSCS-ART when $n < 5000$. Otherwise, RBCVT-Fast will be faster than FSCS-ART. However, its overhead is not so light, and is always about 100 times to that of DF-FSCS algorithm. In Figure 4(b), we can find that DF-FSCS is also faster than two basic forgetting algorithms (i.e., RF-FSCS and CR-FSCS), and its computational overhead is only about one third of theirs. Meanwhile, we notice that the overhead of DF-FSCS will increase in a wave-like form. Two obvious singular points

1) It is available at www.steam.ualberta.ca/main/Papers/RBCVT.

Table 1 The overhead comparison on DF-FSCS and other ARTs for selecting test cases

Algorithm	Computation time (s)						
	$n = 100$	$n = 500$	$n = 1000$	$n = 2000$	$n = 5000$	$n = 10000$	$n = 20000$
DF-FSCS	0.0005	0.0032	0.0067	0.0139	0.0357	0.0807	0.1549
FSCS-ART	0.0018	0.0438	0.1745	0.7025	4.5259	17.9833	68.6238
RBCVT-Fast	0.0741	0.3913	0.7974	1.6310	4.2643	8.8784	18.5912
RF-FSCS ($m = 45$)	0.0021	0.0131	0.2690	0.0546	0.1379	0.2772	0.5583
CR-FSCS ($m = 45$)	0.0019	0.1143	0.2339	0.0473	0.1190	0.2383	0.4779

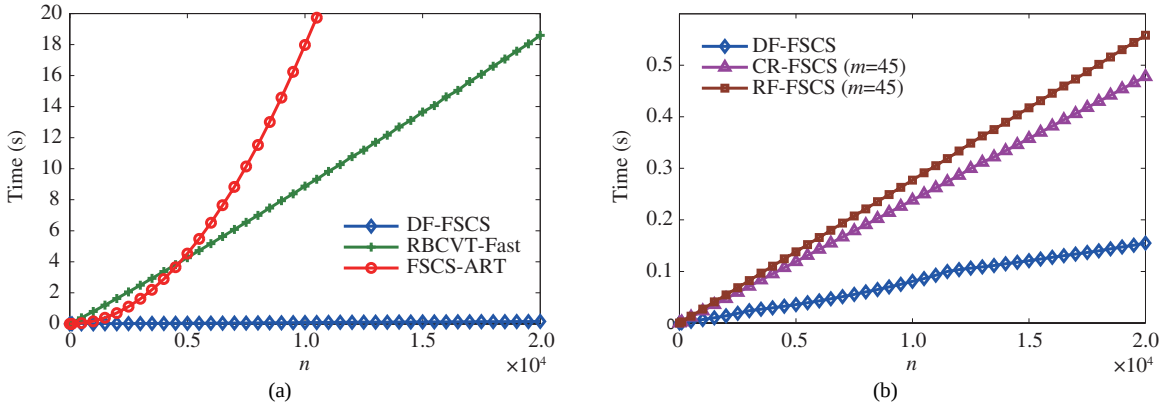

Figure 4 (Color online) The overhead comparison on five algorithms (n from 0 to 2×10^4). (a) DF-FSCS vs. FSCS-ART vs. RBCVT-Fast; (b) DF-FSCS vs. RF-FSCS vs. CR-FSCS.

appear when n is nearby 3000 and 11500, respectively. Nevertheless, this phenomenon is consistent with the theoretical analysis on the complexity of DF-FSCS in Subsection 4.2.

Based on the above observations, we can conclude that DF-FSCS has obvious advantages in computational cost over the FSCS-ART, RBCVT-Fast and two basic forgetting algorithms.

5.3 Analysis on failure-detection effectiveness

At this point, we found overhead reduction achieved by the forgetting strategy of DF-FSCS. However, does the "forgetting" strategy affect the failure detection capability of DF-FSCS? We proposed three research questions to answer it. We aim to find out how DF-FSCS performs as compared with FSCS-ART in RQ2, two forgetting strategies (RF-FSCS and CR-FSCS) in RQ3 and RBCVT-Fast (an ART method with linear-order overhead) in RQ4. Since all above algorithms except RBCVT-Fast can realize the incremental selection of test cases, we answer RQ2 and RQ3 using the metric of F-ratio. By contrast, RBCVT-Fast can only select a fixed-size test suite and hence we answer RQ4 using the metric of P-measure.

5.3.1 *DF-FSCS vs. FSCS-ART*

Since DF-FSCS enhances FSCS-ART by ignoring test cases out of the "sight" of a candidate, here we have to investigate whether it brings an considerable degradation in contrast with FSCS-ART on the failure detection effectiveness.

RQ2: Is the failure-detection effectiveness of DF-FSCS still as good as that of FSCS-ART?

The average F-ratio values of DF-FSCS and FSCS-ART for various cases of failure rate are listed in Table 2. Similar to FSCS-ART algorithm, DF-FSCS has the best performance in block failure pattern, strip pattern comes second, and point pattern is the worst. In block pattern, on the whole, F-ratio of DF-FSCS keeps its values at around 64%. In strip pattern, F-ratio will increase from 92.07% to 98.68% when θ drops from 0.01 to 0.0001. In point pattern, DF-FSCS's F-ratio value is about 98%, and decreases

Table 2 Comparison analysis between DF-FSCS and FSCS-ART on F-ratio

Failure rate	Block pattern		Strip pattern		Point pattern	
	DF-FSCS (%)	FSCS-ART (%)	DF-FSCS (%)	FSCS-ART (%)	DF-FSCS (%)	FSCS-ART (%)
0.01	68.51	68.78	92.07	93.14	99.66	100.63
0.005	65.55	65.76	93.92	93.75	99.20	99.68
0.002	64.60	64.15	95.90	96.39	98.93	97.46
0.001	63.85	63.36	96.58	97.23	98.20	98.96
0.0005	64.34	63.42	98.07	97.22	98.38	97.89
0.0002	62.41	62.55	97.94	98.24	96.01	97.66
0.0001	62.57	61.84	98.68	99.32	97.42	97.18

Table 3 The ANOVA test on F-ratio between DF-FSCS and FSCS-ART at the 0.05 significant level, where Mean Diff. $(x - y)$ means the difference between the average of DF-FSCS's F-ratio (x) and the average of FSCS-ART's F-ratio (y)

Failure rate	Block pattern		Strip pattern		Point pattern	
	Mean Diff. $(x - y)$ (%)	p-value	Mean Diff. $(x - y)$ (%)	p-value	Mean Diff. $(x - y)$ (%)	p-value
0.01	−0.27	0.7042	−1.07	0.3827	−0.97	0.4811
0.005	−0.21	0.7628	0.17	0.8971	−0.48	0.7188
0.002	0.45	0.5173	−0.49	0.7132	1.47	0.2776
0.001	0.49	0.4713	−0.65	0.6334	−0.76	0.5786
0.0005	0.92	0.1898	0.85	0.5381	0.49	0.7165
0.0002	−0.14	0.8450	−0.30	0.8303	−1.65	0.2161
0.0001	0.73	0.2822	−0.64	0.6419	0.24	0.8585

with the decrease of θ value. It should be noted that, the changing trend of F-ratio of DF-FSCS in each failure pattern is very consistent with that of FSCS-ART.

For the comparison between DF-FSCS and FSCS-ART, we performed ANOVA (ANalysis Of VAriance) test on the results of these two algorithms. For block failure pattern, as shown in Table 3, the average F-ratio of DF-FSCS is slightly lower than that of FSCS-ART in cases of $\theta = 0.01$, 0.005 and 0.0002, and opposite in other cases. More importantly, the p-values indicate that DF-FSCS and FSCS-ART have no significant difference in failure-detection capability. For other two patterns, i.e. strip and point patterns, the results are similar to the case of block pattern: the p-values in all cases are far higher than 0.05. Therefore, we can conclude that DF-FSCS and FSCS-ART are comparable (i.e., no significant difference) from the perspective of failure-detection effectiveness.

Based on the above analysis, we can answer question RQ2 that DF-FSCS has no significant difference as compared with FSCS-ART with respect to failure-detection effectiveness although it adopts approximate distance computation.

5.3.2 *DF-FSCS vs. basic forgetting strategies*

Here, we further investigate the difference on failure-detection effectiveness between DF-FSCS and two basic forgetting algorithms (i.e. RF-FSCS and CR-FSCS).

RQ3: Does DF-FSCS have better performance than RF-FSCS and CR-FSCS to detect failures?

As shown in Table 4, the F-ratio of DF-FSCS is always lower than those of basic forgetting algorithms regardless of failure patterns. But, the degrees of difference are not the same for different patterns. In block pattern, compared with both RF-FSCS and CR-FSCS, DF-FSCS always exhibits significant superiority at each category of failure rate. In strip pattern, the difference gradually decreases with the decrease in failure rates. Specifically, when θ changes from 0.01 to 0.0002, there is a significant performance enhancement of DF-FSCS over the two basic forgetting algorithms. However, when θ drops to 0.0001, the difference is not significant, because p-values between DF-FSCS and two basic forgetting algorithms are 0.0574 and 0.2686, respectively. In point pattern, although F-ratio of DF-FSCS is still lower than those of RF-FSCS and CR-FSCS, the difference is not always significant. Meanwhile, with the variance of failure rate, the differences in this failure pattern do not show any obvious regularity.

Table 4 The ANOVA test on F-ratio between DF-FSCS (x) and two basic forgetting strategies ($z1$ and $z2$) at the 0.05 significant level

Failure pattern	Failure rate	RF-FSCS ($m = 45$, $z1$)		CR-FSCS ($m = 45$, $z2$)	
		Mean Diff. ($x - z1$) (%)	p-value	Mean Diff. ($x - z2$) (%)	p-value
Block	0.01	−25.85	0.0000	−16.45	0.0000
	0.005	−40.05	0.0000	−31.13	0.0000
	0.002	−43.10	0.0000	−42.46	0.0000
	0.001	−45.00	0.0000	−41.95	0.0000
	0.0005	−45.85	0.0000	−44.95	0.0000
	0.0002	−44.25	0.0000	−41.86	0.0000
	0.0001	−44.21	0.0000	−45.00	0.0000
Strip	0.01	−8.20	0.0000	−5.61	0.0000
	0.005	−4.70	0.0004	−5.96	0.0000
	0.002	−4.88	0.0004	−6.70	0.0000
	0.001	−4.69	0.0007	−4.02	0.0044
	0.0005	−5.02	0.0005	−3.44	0.0152
	0.0002	−2.97	0.0370	−3.29	0.0202
	0.0001	−2.65	0.0574	−1.55	0.2686
Point	0.01	−6.24	0.0000	−6.65	0.0000
	0.005	−3.27	0.0189	−2.77	0.0467
	0.002	−3.89	0.0058	−1.19	0.3939
	0.001	−1.49	0.2826	−5.63	0.0000
	0.0005	−1.60	0.2510	−0.59	0.6701
	0.0002	−3.97	0.0043	−3.25	0.0191
	0.0001	−1.95	0.1526	−2.29	0.0987

In summary, in terms of failure-detection capability, DF-FSCS significantly outperforms RF-FSCS and CR-FSCS for block failure pattern. This outperformance also occurs in the case of relatively high failure rate (i.e. $\theta > 0.0001$) for strip pattern. In other cases, the difference is not very significant, but the mean of DF-FSCS's F-ratio values is still lower than those of two basic forgetting algorithms.

5.3.3 *DF-FSCS vs. RBCVT-Fast*

It has been reported that RBCVT-Fast's P-measure outperforms the standard FSCS-ART algorithm in most cases [14]. Here, we investigate the research question as follows.

RQ4: Is DF-FSCS's P-measure comparable to that of RBCVT-Fast?

For fair comparisons, the sizes of test suites at each category of failure rates were set the same as those in the study of Shahbazi et al. [14], that is, $n = 69$, 693 and 6931 for $\theta = 0.01$, 0.001 and 0.0001, respectively. The corresponding P-measure values and ANOVA test results are shown in Table 5. In block pattern, the P-measure of RBCVT-Fast algorithm is always higher than that of DF-FSCS, and the difference between them is significant (i.e. p-value < 0.05). From the aspect of change trend, the P-measure of DF-FSCS gradually increases with the decrease in failure rate. However, the P-measure of RBCVT-Fast has the opposite trend. Hence, the difference of these two algorithms diminishes with the decrease in failure rate. In strip pattern, the failure-detection capabilities of these two algorithms are comparable. When $\theta = 0.01$, the P-measure of RBCVT-Fast surpasses that of DF-FSCS slightly. For other two cases, DF-FSCS shows a slight performance over RBCVT-Fast. But, in all three cases, the differences between them are not significant. In point pattern, when failure rate is high (e.g., $\theta = 0.01$), RBCVT-Fast is significantly better than DF-FSCS. Otherwise, they are comparable.

That is to say, RBCVT-Fast outperforms DF-FSCS in block pattern and the high failure rate cases in point pattern. In other cases, they are comparable. Although RBCVT-Fast shows some advantages over DF-FSCS in P-measure, RBCVT-Fast has its inherent limitation: it is applicable for a pre-defined size

Table 5 The ANOVA test on P-measure between DF-FSCS (x) and RBCVT-Fast (w) at the 0.05 significant level

Failure pattern	Failure rate	n	DF-FSCS (x)	RBCVT-Fast (w)	Mean Diff. ($x-w$)	p-value
Block	0.01	69	0.5799	0.7465	−0.1666	0.0000
	0.001	693	0.6178	0.7068	−0.0890	0.0000
	0.0001	6931	0.6298	0.6896	−0.0598	0.0000
Strip	0.01	69	0.5193	0.5249	−0.0056	0.4355
	0.001	693	0.5082	0.4964	0.0118	0.1103
	0.0001	6931	0.5046	0.5033	0.0013	0.8650
Point	0.01	69	0.4954	0.5245	−0.0291	0.0000
	0.001	693	0.5030	0.5240	−0.0210	0.0043
	0.0001	6931	0.5046	0.5112	−0.0066	0.3493

of test suite. By contrast, DF-FSCS can realize the incremental selection of test cases, and it is faster than RBCVT-Fast about 100 times.

5.4 Discussion

Based on the simulation analysis, we can confirm that DF-FSCS algorithm has fulfilled the purpose of reduction on computational overhead. It reduces the time complexity of FSCS-ART from $O(n^2)$ to $O(n)$. Even compared to the two basic forgetting methods and RBCVT-Fast, it still exhibits obvious advantage in efficiency.

Although DF-FSCS ignores considerable test cases for distance computation, its capability of detecting failures is still comparable to FSCS-ART. Compared with two basic forgetting methods, DF-FSCS significantly outperforms them for the block failure pattern and most cases of the strip pattern, and has better performance but not very significant for the point pattern. On the other hand, DF-FSCS has comparable failure-detection effectiveness as the typical linear-time ART algorithm RBCVT-Fast in the strip pattern and the low failure rate cases of the point pattern, but worse in other cases.

6 Empirical analysis

6.1 Subject programs and experimental setup

In this section, we further validate the effectiveness of DF-FSCS algorithm through employing empirical studies on 12 real programs that have been extensively used in ART studies [2, 21]. They are all open published programs from ACM's collected algorithms [22] and the Numerical Recipes [23]. In our experiments, we converted the C++ versions in [2] to Java programs. The details of these subject programs, such as input domain, number of seeded faults and failure rate, are listed in Table 6. The dimension numbers of these programs vary from 1 to 4. Apart from programs `bessj`, `gammq`, `plgndr` and `cel`, the input domains of other programs are equilateral, that is the dimension lengthes of program's input domain are equal to each other.

The failure rate of each faulty program in Table 6 was calculated as follows: at the preparation stage of empirical analysis, a huge amount of test inputs were successively picked up from input domain by taking a small value as step size, and then they were used to execute program and observe program behaviors (i.e., pass or fail). Accordingly, failure rate θ could be calculated as the ratio of fail times to the number of all test inputs. All twelve programs were seeded with a number of faults in a manual way. The faults belong to different types of the common mutant operations [24], including arithmetic operator replacement (AOR), relational operator replacement (ROR), scalar variable replacement (SVR), constant replacement (CR) and other (OTH). Accordingly, their failure rates lie in the approximate range from 0.0003 to 0.002.

In fact, a certain number of mutant program versions have been generated at the preparation stage. Since RT methods can easily detect the first fault when the program has a high failure rate, this kind

Table 6 Subject programs for empirical study

Program	d	Input domain		Total faults	Failure rate (θ)	Cells in initial partition (φ_0)
		From	To			
airy	1	−5000	5000	1	0.000716	3
bessj0	1	−300000	300000	5	0.001373	3
erfcc	1	−30000	30000	4	0.000574	3
probks	1	−50000	50000	4	0.000387	3
tanh	1	−500	500	4	0.001817	3
bessj	2	(2, −1000)	(300, 15000)	4	0.000716	1×54
gammq	2	(0, 0)	(1700, 40)	4	0.000830	43×1
sncndn	2	(−5000, −5000)	(5000, 5000)	5	0.001623	3×3
golden	3	(−100, −100, −100)	(60, 60, 60)	5	0.000550	$3 \times 3 \times 3$
plgndr	3	(10, 0, 0)	(500, 11, 1)	5	0.000368	$490 \times 11 \times 1$
cel	4	(0.001, 0.001, 0.001, 0.001)	(1, 300, 10000, 1000)	3	0.000332	$1 \times 1 \times 34 \times 4$
el2	4	(0, 0, 0, 0)	(250, 250, 250, 250)	9	0.000690	$3 \times 3 \times 3 \times 3$

of programs is not suitable to clearly distinguish which RT/ART testing method is really effective. Therefore, the faulty programs with a relatively high failure rate were omitted in the experiments. That is to say, only the programs whose failure rate is less than 0.002 were taken into consideration. Although fault types and fault numbers may have an impact on the failure-detection effectiveness, their impact is reflected by the failure rate, on which the effectiveness of random testing heavily depends. Moreover, the failure rate is a key indicator of the difficulty level in which faults are detected. Hence, we took it as the important feature to describe the faults in programs here. Take the program `airy` as an example, only one fault is seeded into this program, but its failure rate is a fairly low value (i.e. 0.000716), so this mutant version is suitable for comparative analysis.

In DF-FSCS algorithm, it needs to provide an initial partition for the input domain of each program. if the range of all dimensions of the input domain are of equal lengths, we divide each dimension into $p_0 = 3$ parts. Otherwise, we use the shortest side ($l_{\min}$) of the input domain as the basic side of cell (l_{side}) to perform partitioning. In particular, if the ratio of the longest side ($l_{\max}$) of the input domain to the shortest side is very large (e.g., it is about 10000 for program `cel`), the second shortest side can be used as l_{side} for partitioning so as to control the number of initial cells. According to Definition 2, in d-dimensional space, the upper bound of the number of cells within a neighbour region is 3^d. To compare with basic forgetting methods, the memory size (denoted as m) in RF-FSCS or CR-FSCS for d-dimensional space should be $\tau \cdot 3^d$. When d varies from 1 to 4, m is set as 15, 45, 135 and 405, respectively. In addition, the running environment is the same as the former simulation analysis.

In ART, the selection time of test cases is mainly determined by the number of test cases rather than program structure. Thus, the conclusion on computation efficiency in Subsection 5.2 is also suitable to real-world programs. For this reason, we did not repeat the experiments on comparing computation time here, but only focused on the failure-detection effectiveness.

6.2 Comparison analysis on F-measure

In order to investigate the failure-detection capability of DF-FSCS with respect to F-measure, the following research question should be studied.

RQ5: For real programs, is the failure-detection capability of DF-FSCS comparable to that of FSCS-ART, and better than those of RF-FSCS and CR-FSCS?

The comparison between DF-FSCS and FSCS-ART is listed in Table 7. For 12 real programs, the F-measure of DF-FSCS is lower than that of FSCS-ART in most cases. However, the difference between them is not very significant (p-value > 0.05). That is to say, the failure-detection capabilities of DF-FSCS and FSCS-ART are comparable for these 12 programs.

Further, we also performed comparison analysis on DF-FSCS and other two basic forgetting methods,

Table 7 The F-measure results of DF-FSCS (x) and FSCS-ART (y) and ANOVA test on them at the 0.05 significant level

Program	DF-FSCS (x)	FSCS-ART (y)	Mean Diff. ($x-y$)	Significance
airy	793.56	803.86	−10.3	0.3500
bessj0	437.28	450.65	−13.37	0.2918
erfcc	2858.48	2898.70	−40.22	0.3107
probks	5594.51	5565.88	28.63	0.7149
tanh	311.17	316.45	−5.28	0.2224
bessj	436.96	437.43	−0.47	0.9513
gammq	1054.38	1059.96	−5.58	0.7848
sncndn	622.57	625.53	−2.96	0.8159
golden	1582.05	1580.13	1.92	0.9512
plgndr	1665.35	1624.84	40.51	0.1566
cel	1525.35	1565.42	−40.07	0.1782
el2	702.08	704.71	−2.63	0.8518

Table 8 The ANOVA test on F-measure between DF-FSCS (x) and two basic forgetting methods ($z1$ and $z2$) for real programs

Program	RF-FSCS ($z1$)		CR-FSCS ($z2$)	
	Mean Diff. ($x-z1$)	Significance	Mean Diff. ($x-z2$)	Significance
airy	−680.99	1.4263×10^{-195}	−662.50	8.1218×10^{-185}
bessj0	−350.31	3.2171×10^{-178}	−339.53	4.2733×10^{-172}
erfcc	−2452.54	3.5252×10^{-192}	−2316.34	2.1944×10^{-188}
probks	−4349.88	1.7722×10^{-183}	−4205.00	1.5192×10^{-181}
tanh	−255.69	1.9507×10^{-180}	−248.46	1.5656×10^{-181}
bessj	−239.09	1.7028×10^{-99}	−118.74	2.6070×10^{-36}
gammq	177.24	3.7859×10^{-21}	44.98	0.0286
sncndn	−16.26	0.1980	−12.63	0.3168
golden	−17.76	0.5736	3.10	0.9211
plgndr	502.60	3.8085×10^{-83}	491.30	2.3138×10^{-80}
cel	−244.77	4.8202×10^{-14}	−165.65	1.6231×10^{-07}
el2	−19.25	0.1729	1.16	0.9334

and the corresponding results are shown in Table 8. In the case of $d=1$, DF-FSCS is significantly better than RF-FSCS and CR-FSCS for all five programs. In the case of $d=2$, the F-measure of DF-FSCS is lower than those of RF-FSCS and CR-FSCS for programs `bessj` and `sncndn`, but the difference is significant only for `bessj`. In particular, RF-FSCS and CR-FSCS show better performance than DF-FSCS for program `gammq`, and even better than FSCS-ART. In the case of $d=3$, for program `golden`, the F-measure of DF-FSCS is lower than that of RF-FSCS, and higher than that of CR-FSCS, but the differences in both comparisons are not significant. For program `plgndr`, similar to the result of program `gammq`, the basic forgetting methods have better performance than both DF-FSCS and FSCS-ART. In the case of $d=4$, DF-FSCS is significantly better than other two methods for program `cel`, but has no obvious outperformance for program `el2`.

Based on the above analysis, we can conclude that DF-FSCS's performance is comparable to FSCS-ART for all 12 programs. It is significantly better than two basic forgetting methods for programs with 1 or 2-dimensional input domain, but is comparable for the other cases.

6.3 Comparison analysis on P-measure

Since RBCVT-Fast algorithm is mainly used for generating fixed-size test suite, P-measure is an appropriate metric to describe its effectiveness. Hence, a comparison between DF-FSCS and RBCVT-Fast on

such a metric is investigated here.

RQ6: Does RBCVT-Fast always have higher P-measure than those of DF-FSCS, FSCS-ART and two basic forgetting strategies?

According to the description in RBCVT-Fast algorithm, the parameter α represents the ratio of test cases in a basic cell in the border outside input domain (denoted as c_H) to test cases in a cell within input space (denoted as c_I), here c_H and c_I are the basic cells with the same side length in each dimension, so their areas (or volumes) are identical. For 1-dimensional (1-D) and 2-dimensional (2-D) input domain, the recommended value $\alpha = 2.0$ from reference [14] is suitable. However, if α is kept unchanged for programs with 3-D or 4-D input domain, test cases will hardly have chance to enter the border of input domain. Thus, we assigned α to 1.0 for 3-D and 4-D input domain in our studies.

The P-measure results of five algorithms for all 12 programs are shown in Figure 5. For five programs with 1-D input domain, DF-FSCS, RBCVT-Fast and FSCS-ART produce nearly identical results in most test suite sizes (i.e., n). But, the P-measure values of two basic forgetting methods are always lower than those of the above three algorithms. In the case of $d = 2$, DF-FSCS still achieves the same P-measure results as FSCS-ART for all three programs. Compared to RBCVT-Fast algorithms, DF-FSCS and FSCS-ART always have higher P-measure for programs `bessj` (Figure 5(f)) and `gammq` (Figure 5(g)). For program `bessj`, the result of CR-FSCS is better than that of RF-FSCS, and they are both worse than DF-FSCS and FSCS-ART. Instead, these two basic forgetting methods outperforms three other algorithms for the program `gammq`. For program `sncndn`, all five algorithms have no obvious difference on metric P-measure irrespective of the test suite size.

In the case of $d = 3$, for program `golden`, the P-measure results of DF-FSCS, FSCS-ART and two basic forgetting methods are nearly the same. RBCVT-Fast outperforms DF-FSCS in most cases of size n, especially in the following two intervals: $[600, 1000]$ and $[1400, 2000]$. For program `plgndr`, RBCVT-Fast is always better than the other four algorithms. Meanwhile, P-measure values of RF-FSCS and CR-FSCS are nearly identical, and they are higher than those of FSCS-ART and DF-FSCS. Further, DF-FSCS is worse than FSCS-ART when the number of test cases is less than 1000.

In the case of $d = 4$, for program `cel`, P-measure of RBCVT-Fast is almost equal to 1.0 in all cases of size n, so it is always better than other four algorithms. On the contrary, RBCVT-Fast is worse than other algorithms for program `el2`. For both programs with 4-D input domain, DF-FSCS has the same effect to algorithm FSCS-ART. Only when $n > 800$ for program `cel`, two basic forgetting methods are worse than DF-FSCS and FSCS-ART, otherwise, these four algorithms have no significant difference.

Based on the above observations, we can now answer RQ6 as below: (1) DF-FSCS has nearly identical performance on P-measure to FSCS-ART algorithm, with the exception of the case $n < 1000$ for program `plgndr`. (2) For about half of subject programs in our studies, RBCVT-Fast has no obvious difference to DF-FSCS and FSCS-ART. For programs `golden`, `plgndr` and `cel`, RBCVT-Fast shows better performance than DF-FSCS and FSCS-ART, but the reverse observations are for programs `bessj`, `gammq` and `el2`. (3) In most cases, two basic forgetting methods' performance is worse than or equal to DF-FSCS and FSCS-ART. (4) With the growth of input domain's dimension, the performance of RBCVT-Fast becomes instable.

On the other hand, we find that the value of parameter α in RBCVT-Fast has an impact on the probability of test cases lying on (or closing to) the boundary of input domain. Specifically, a higher α value generally means that test case has less chance of staying at the border area of input domain. In order to investigate this phenomenon, we took programs with 4-dimensional inputs for further investigations and varied α value to observe the change of P-measure. In the experiments, α varies from 1.0 to 1.4 for program `cel`, and from 0.6 to 1.0 for program `el2`. According to the experimental results in Figure 6, the sensitivity of RBCVT-Fast's performance to α value has been verified again. As shown in Figure 6(a), RBCVT-Fast's P-measure for program `cel` is always close to 1.0 when α is equal to 1.0. However, when α increases to 1.2, its performance is obviously degraded, and the P-measure is dropped to about 0.2 when α reaches 1.4. At the same time, the fluctuation of P-measure is also very frequent in the cases of $\alpha = 1.2$ or 1.4. For program `el2` (Figure 6(b)), α value also has an obvious influence on the P-measure of RBCVT-Fast algorithm. For all three values of α, the P-measure has no steady and consistent relation

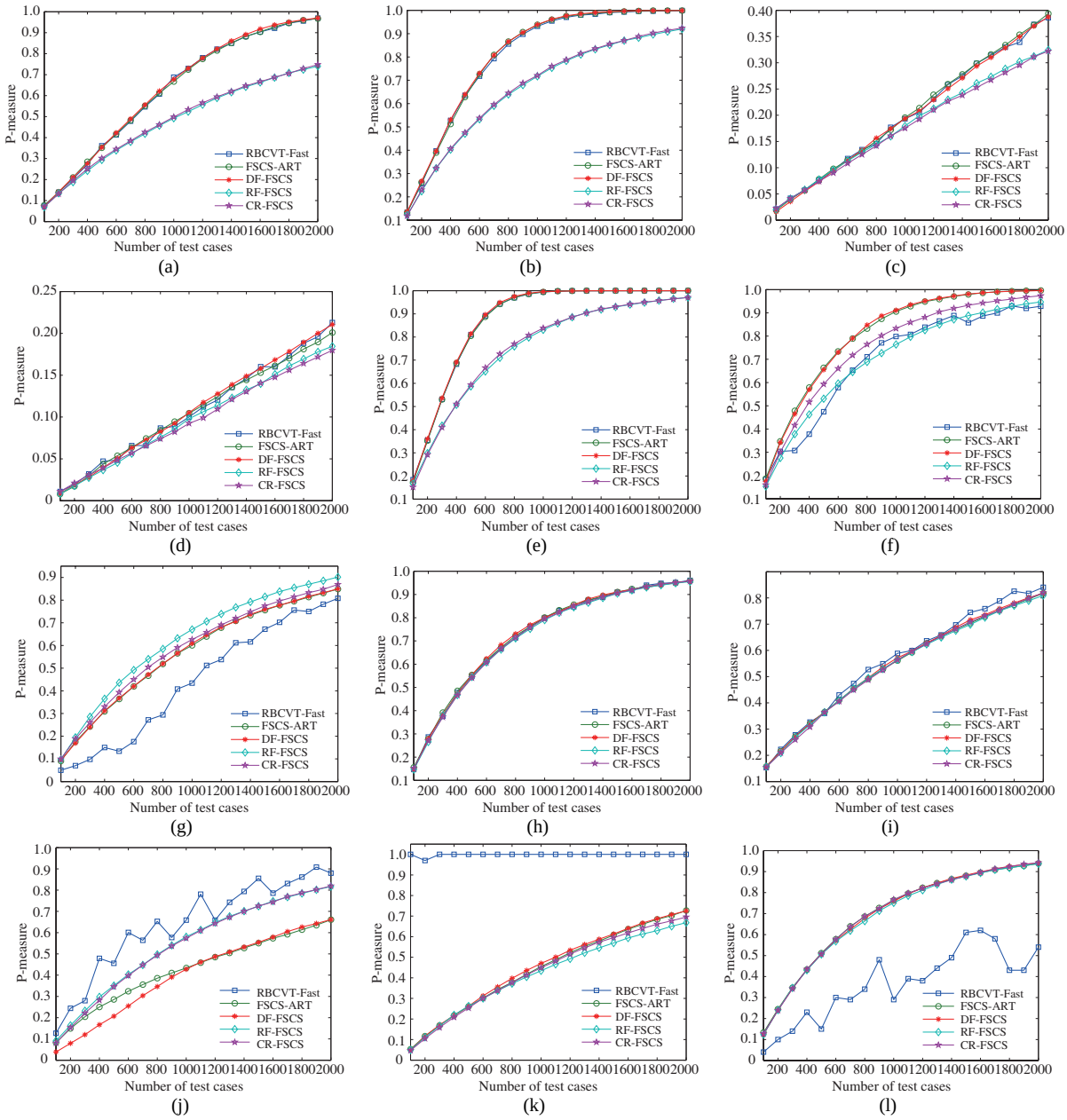

Figure 5 (Color online) The P-measure results of all 12 programs. (a) airy ($d = 1$); (b) bessj0 ($d = 1$); (c) erfcc ($d = 1$); (d) probks ($d = 1$); (e) tanh ($d = 1$); (f) bessj ($d = 2$); (g) gammq ($d = 2$); (h) sncndn ($d = 2$); (i) golden ($d = 3$); (j) plgndr ($d = 3$); (k) cel ($d = 4$); (l) el2 ($d = 4$).

with the growth of test case number.

Through the above preliminary analysis, we have found another possible drawback of RBCVT-Fast algorithm: when the dimension of input domain increases to a certain level, such as $d = 3$ or $d = 4$ in our empirical studies, the failure-detection capability of RBCVT-Fast heavily depends on the setting of parameter α. If α is not properly set, the performance of algorithm will be greatly reduced. However, for testers, there is no other information available for setting an appropriate values for α, so it is very difficult to configure an optimal settings for RBCVT-Fast to select test cases.

Finally, based on the results of simulation and empirical analyses, the application guidelines for our DF-FSCS algorithm and RBCVT-Fast are as below: (1) when testers have adequate information about the failure rate of programs under test or knowledge of the testing resources to determine the size of test suite, RBCVT-Fast is the preferred algorithm for adaptive random testing. Otherwise, DF-FSCS

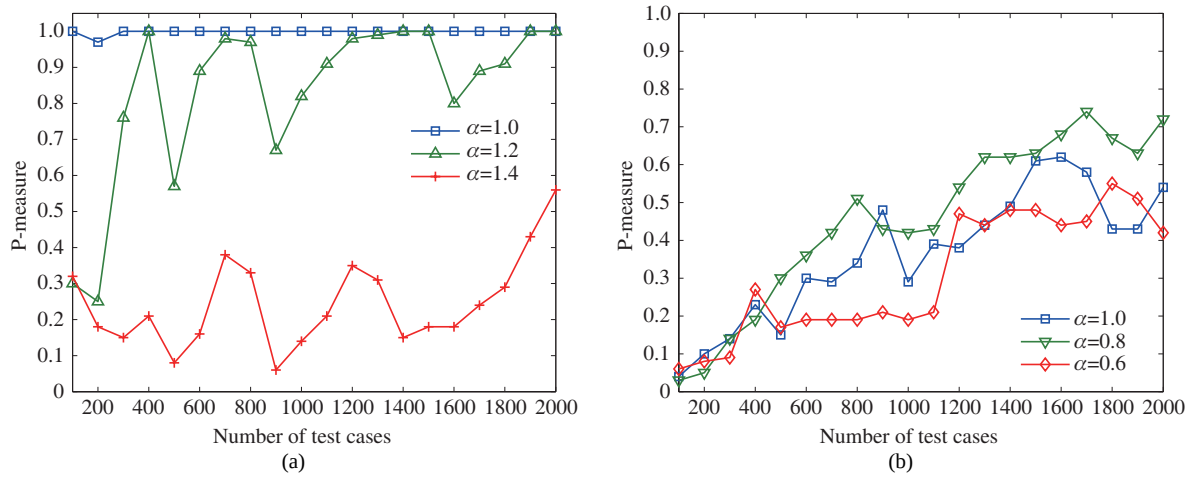

Figure 6 (Color online) The P-measure results in different values of parameter α in RBCVT-Fast algorithm. (a) cel ($d = 4$); (b) el2 ($d = 4$).

is recommended. (2) If the program execution time is very short, the faster algorithm of selecting test cases, i.e., our algorithm DF-FSCS might be a more preferable choice. By contrast, if the program takes a long time to execute, the selection time of test cases will not bring a remarkable impact on program testing processes, so both of them are suitable.

7 Related work

In the paper, we attempt to reduce the computational overhead through forgetting the test cases outside the neighbour region of each candidate. Here, we briefly review some current techniques closely related to this work.

The overhead of FSCS-ART has been concerned for a long time, and a few methods have been proposed to address it. Typically, the technique of mirroring is proposed to address this problem [25]. Although it can save distance computation overhead, its complexity is still of the order of n^2. On the other hand, our DF-FSCS algorithm can reduce the time complexity of FSCS-ART into linear order without loss of failure-detection capability. Subsequently, the forgetting strategy is introduced to reduce the overhead of distance computation in ART [12]. Unfortunately, it is difficult to balance between efficiency and effectiveness: the small size of memory set can effectively reduce the computational overhead, but will lead to a low failure-detection capability, and vice versa. The primary cause of the above dilemma is that the basic forgetting strategies do not consider the spatial distribution of test cases.

Besides the selection-based approach [5], e.g. FSCS-ART, partitioning is also an important approach to select test cases from input domain. The typical methods include random partitioning, bisection [26], grid partitioning [27, 28]. Although the concepts of occupied cell and adjacent cell have been utilized in IP-ART algorithm [27], the main purpose is to identify the available cells (i.e., the cells have not been occupied and are also not adjacent cells of the occupied ones) from which next test cases can be selected. In our approach, we use the neighbour region to exclude the unnecessary test cases in TS for distance computation. In essence, IP-ART belongs to exclusion-based ART, whereas our approach still belongs to the selection-based ART [5, 21].

In RBCVT-Fast algorithm, grid partitioning is also used to reduce the cost to compute the centroid of background points [14]. However, the neighbour region in RBCVT-Fast gradually expands outwards according to the measure of Chebyshev distance. The number of adjacent cells does not have a fixed upper-bound. Accordingly, the number of test cases used for distance computation (i.e., the test cases lie in the adjacent cells) does not have a specific upper-bound in theory. By contrast, our algorithm restricts the neighbour region of a point to 3^d cells around it. Further, we use a second-round of forgetting to control the number of test cases within the neighbour region. Therefore, the overhead of distance computation

for each candidate is a constant. On the other hand, RBCVT-Fast algorithm incurs quite a considerable amount of background points to relocate test cases, and clustering action has to be performed frequently on these points. Therefore, its computational overhead is not light, though its complexity in theory is linear. Due to the use of Centroidal Voronoi Tessellations (CVT), RBCVT-Fast algorithm requires the number of test cases to be fixed prior to testing. In practice, it may be difficult for a tester to know how many test cases are required in advance, especially in the lack of information about failure rate. In addition, the failure-detection capability of RBCVT-Fast will be unstable and depends on the setting of parameter α, especially in the case of high-dimensional input domain.

Chow et al. [28] proposed an efficient ART algorithm by applying divide and conquer technique to partition input domain, and then use FSCS-ART to select test cases in each sub-domain. Their approach is effectively an integration of the partitioning-based ART and selection-based ART. Different from their work, DF-FSCS does not conduct FSCS-ART separately in different sub-domains, the partitioning scheme is only used to determine the neighbour region of each candidate, outside of which already executed test cases can be forgotten. Recently, a linear-time ART algorithm (ARTsum) for software with non-numeric inputs has been proposed by Barus et al. [29]. In their approach, concepts of categories and choices from category-partition testing is used to formulate the distance measure between test cases. Here, we still concern on the testing for programs with numeric inputs, so the Euclidean distance measure is adopted. For this reason, the comparison between their linear-time algorithm and DF-FSCS is not performed because they deal with different types of programs.

8 Conclusion

Adaptive random testing has exhibited a better capability for detecting program failures than random testing. However, since it attempts to ensure an even spread of test cases, the overhead problem accordingly appears. Hence, how to effectively reduce the overhead of ART methods becomes a significant task. To address this problem, we proposed a forgetting strategy from the perspective of spatial distribution of test cases. In our forgetting strategy named DF-FSCS, only the test cases lying within the "sight" (i.e., neighbour region) of a candidate are used in distance computation. In order to control the number of test cases within neighbour region, the dynamic adjustment and the second-round forgetting are applied in DF-FSCS algorithm.

The simulation analysis and empirical study for DF-FSCS and other related ART algorithms are also conducted. The results show that DF-FSCS can significantly reduce the overhead of FSCS-ART without at the expense of the failure-detection capability. Although DF-FSCS has the same worst-case complexity to two basic forgetting strategies, its runtime cost is lower than them in practice. More importantly, the failure-detection effectiveness of DF-FSCS has a significant improvement than them in most cases. Although RBCVT-Fast has better effectiveness in some cases such as the block failure pattern, the experimental results on 12 real-world programs show that DF-FSCS is comparable to RBCVT-Fast in terms of failure-detection effectiveness. On the other hand, DF-FSCS exhibits a significantly lighter (only about 1%) overhead than RBCVT-Fast for selecting test cases. In addition, DF-FSCS is more stable than RBCVT-Fast for different programs, especially in the case of high-dimensional input domain.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 61462030), Australian Research Council Linkage Grant (Grant No. LP100200208), Natural Science Foundation of Jiangxi Province (Grant Nos. 20162BCB23036, 20151BAB207018), and Science Foundation of Jiangxi Educational Committee (Grant No. GJJ150465).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Hamlet R. Random testing. In: Marciniak J J, ed. Encyclopedia of Software Engineering. 2nd ed. Chichester: John Wiley and Sons, 2002. 1507–1513

2 Chen T Y, Leung H, Mak I K. Adaptive random testing. In: Proceedings of the 9th Asian Computing Science Conference, Chiang Mai, 2004. 320–329
3 Chen T Y, Kuo F-C, Merkel R G, et al. Adaptive random testing: the ART of test case diversity. J Syst Softw, 2010, 83: 60–66
4 Orso A, Rothermel G. Software testing: a research travelogue (2000-2014). In: Proceedings of Future of Software Engineering (FOSE'14), Hyderabad, 2014. 117–132
5 Chen T Y, Kuo F-C, Towey D, et al. A revisit of three studies related to random testing. Sci China Inf Sci, 2015, 58: 052104
6 Shi Q, Chen Z, Fang C, et al. Measuring the diversity of a test set with distance entropy. IEEE Trans Reliab, 2016, 65: 19–27
7 Feldt R, Poulding S, Clark D, et al. Test set diameter: quantifying the diversity of sets of test cases. In: Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16), Chicago, 2016. 223–233
8 Chen T Y, Merkel R G. An upper bound on software testing effectiveness. ACM Trans Softw Eng Methodol, 2008, 17: 1–27
9 Chen T Y. Fundamentals of test case selection: diversity, diversity, diversity. In: Proceedings of the 2nd International Conference on Software Engineering and Data Mining (SEDM'10), Chengdu, 2010. 723–724
10 Mariani L, Pezzè M, Zuddas D. Recent advances in automatic black-box testing. Adv Comput, 2015, 99: 157–193
11 Arcuri A, Briand L. Adaptive random testing: an illusion of effectiveness? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), Toronto, 2011. 265–275
12 Chan K P, Chen T Y, Towey D. Forgetting test cases. In: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Chicago, 2006. 485–494
13 Rigaux P, Scholl M, Voisard A. Spatial Databases: With Application to GIS. San Francisco: Morgan Kaufmann Publishers, 2002. 267–309
14 Shahbazi A, Tappenden A F, Miller J. Centroidal voronoi tessellations—a new approach to random testing. IEEE Trans Softw Eng, 2013, 39: 163–183
15 Finelli G B. NASA software failure characterization experiments. Reliab Eng Syst Saf, 1991, 32: 155–169
16 Bishop P G. The variation of software survival time for different operational input profiles. In: Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, 1993. 98–107
17 Schneckenburger C, Mayer J. Towards the determination of typical failure patterns. In: Proceedings of the 4th International Workshop on Software Quality Assurance (SOQUA'07), Dubrovnik, 2007. 90–93
18 Chen T Y, Tse T H, Yu Y T. Proportional sampling strategy: a compendium and some insights. J Syst Softw, 2001, 58: 65–81
19 Chen T Y, Huang D H. Adaptive random testing by localization. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), Busan, 2004. 292–298
20 Chen T Y, Kuo F-C, Merkel R. On the statistical properties of testing effectiveness measures. J Syst Softw, 2006, 79: 591–601
21 Kuo F-C. On adaptive random testing. Dissertation for the Doctoral Degree. Melbourne: Swinburne University of Technology, 2006. 24–26
22 ACM. Collected Algorithms from ACM: Volume 1, Algorithms 1–220. New York: Association for Computer Machinery, 1980
23 Press W H, Teukolsky S A, Vetterling W T, et al. Numerical Recipes: The Art of Scientific Computing. 3rd ed. Cambridge: Cambridge University Press, 2007
24 Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng, 2011, 37: 649–678
25 Chen T Y, Kuo F-C, Merkel R G, et al. Mirror adaptive random testing. Inform Softw Tech, 2004, 46: 1001–1010
26 Chen T Y, Merkel R, Eddy G, et al. Adaptive random testing through dynamic partitioning. In: Proceedings of the 4th International Conference on Quality Software (QSIC'04), Braunschweig, 2004. 79–86
27 Chen T Y, Huang D, Zhou Z. On adaptive random testing through iterative partitioning. J Inf Sci Eng, 2011, 27: 1449–1472
28 Chow C, Chen T Y, Tse T H. The ART of divide and conquer: an innovative approach to improving the efficiency of adaptive random testing. In: Proceedings of the 13th International Conference on Quality Software (QSIC'13), Nanjing, 2013. 268–275
29 Barus A C, Chen T Y, Kuo F-C, et al. A cost-effective random testing method for programs with non-numeric inputs. IEEE Trans Comput, 2016, 65: 3509–3523