Science of Computer Programming 215 (2022) 102743

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A nearest-neighbor divide-and-conquer approach for adaptive random testing

Rubing Huang [a], Weifeng Sun [b,*], Haibo Chen [c], Chenhui Cui [c], Ning Yang [d]

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

ARTICLE INFO

Article history:
Received 19 December 2020
Received in revised form 27 September 2021
Accepted 9 October 2021
Available online 5 November 2021

Keywords:
Adaptive random testing
Computational overhead
Boundary effect problem
Nearest-neighbor
Divide-and-conquer

ABSTRACT

Adaptive Random Testing (ART) aims at enhancing the failure detection capability of *Random Testing* (RT) by evenly distributing test cases over the input domain. Many ART algorithms have been proposed to achieve an even spread of test cases, according to different notations. A well-known ART algorithm, namely *Fixed-Size-Candidate-Set ART* (FSCS-ART), chooses an element as the next test case such that it is farthest away from previously executed test cases. Previous studies have demonstrated that FSCS-ART could achieve good improvements in test effectiveness over RT, but suffers from some drawbacks such as high computational overhead and boundary effect problem. In this paper, we propose an alternative approach to enhance FSCS-ART, namely *Nearest-Neighbor Divide-and-Conquer based ART* (NNDC-ART), which combines the concepts of *nearest-neighbor* and *divide-and-conquer*. It attempts to reduce the computational overhead of test case generation, and also to alleviate the boundary effect problem; while maintaining the advantages of FSCS-ART. The simulation results show that compared with FSCS-ART, NNDC-ART requires much less computational overhead, and also achieves much better test case distribution; while maintaining better or comparable failure-detection effectiveness. Our empirical studies further show that the proposed approach is much more cost-effective than FSCS-ART. In addition, we further compare NNDC-ART against two enhancements of FSCS-ART based on the partitioning strategy, i.e., *ART with Divide-and-Conquer* (ART-DC) and *ART by Bisection and Localization* (ART-B-Loc), and find that our proposed approach can achieve similar or even better testing performances in some scenarios.

1. Introduction

Software testing is one of the important techniques to ensure the quality of the software. *Random Testing* (RT) [1] may be the simplest software testing technique, which can be considered as the first option to apply [2]. Due to its simplicity and high efficiency, RT has been widely applied to many practical systems such as UNIX utility programs [3], Windows NT applications [4], Java Just-In-Time compilers [5], embedded software systems [6], and SQL database systems [7]. Despite its benefits and popularity, RT still faces a lot of discussion and controversy. For example, Myers [8] criticized that RT has low

* Corresponding author.
E-mail addresses: rbhuang@must.edu.mo (R. Huang), weifeng.sun@cqu.edu.cn (W. Sun), 2221808034@stmail.ujs.edu.cn (H. Chen), 2211908012@stmail.ujs.edu.cn (C. Cui), yangn@ujs.edu.cn (N. Yang).

https://doi.org/10.1016/j.scico.2021.102743
0167-6423/

effectiveness in detecting failures for using little or no information about previously executed test cases to select the next test case.

Many approaches have been proposed to enhance testing effectiveness of RT, especially for failure detection, by introducing additional information to guide test case generation. *Adaptive Random Testing* (ART) [9] is an enhancement of RT based on the knowledge about the contiguity of *failure regions*,[1] which aims at achieving an even spread of random test cases over the input domain. More specifically, previous studies have identified that the failure-causing inputs usually cluster into one or a few contiguous failure regions, and hence non-failure regions should also be contiguous [10–14]. In other words, if a test input *tc* can trigger a software failure (i.e., *tc* is a failure-causing input), the neighbors around *tc* are likely to identify this failure. Similarly, if *tc* is not failure-causing, *tc*'s neighbors are not failure-causing as well. As a result, it is intuitive that a test input that is far away from non-failure-causing test inputs may have a higher chance of causing failure than the neighboring test inputs.

Many ART algorithms have been proposed to achieve an even spread of random test cases over the input domain, in terms of different notations [9]. The most well-known one is the first implementation of ART, namely *Fixed-Size-Candidate-Set ART* (FSCS-ART) [15]. It first selects a fixed number of candidate test cases from the input domain in a random manner, and then chooses an element from candidates as the next test case such that it is farthest away from previously executed (or selected) test cases. Previous studies have demonstrated that FSCS-ART is more effective than RT from the perspective of F-measure (i.e., the expected number of test cases required to detect the first failure) [15–20]. However, FSCS-ART suffers from the following two main weaknesses: *boundary effect problem* and *high computational overhead* [9]. More specifically, Chen et al. [21] pointed out that FSCS-ART suffers from the boundary effect problem, i.e., its candidates close to the center of the input domain have a lower probability to be selected as test cases than those close to the boundary. In other words, with the increase of the number of test cases, more test cases are located near the boundary of the input domain. Additionally, FSCS-ART also faces the challenge of high computational overhead [15,22], because its time complexity order is $O(N^2)$, where N is the number of generated test cases.

Attempts have been made to address the high computational overhead of adaptive random testing, such as *ART by Bisection* (abbreviated as ART-B [23]). ART-B inspired by partition testing does not require distance computations to achieve an even distribution of the test cases. Despite that, its performance measures in terms of F-measure are not nearly as good as the FSCS-ART, especially for low dimensional spaces, where ART-B is 10% worse in some scenarios. As for the boundary effect problem, many algorithms [21,24] have been proposed to solve this challenge. However, to achieve this goal, these approaches introduce more or less additional distance calculation and thus aggravate the computational overhead.

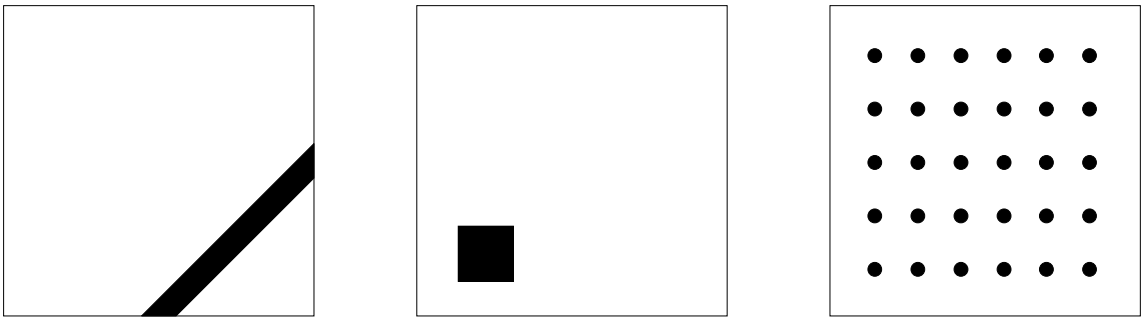
To overcome or alleviate the above two weaknesses of FSCS-ART, in this paper we propose an alternative ART approach by combining the concepts of *nearest-neighbor* and *divide-and-conquer*, namely *Nearest-Neighbor Divide-and-Conquer based ART* (NNDC-ART). Actually, a high computational overhead of FSCS-ART is required to identify the nearest neighbor for each candidate. As a result, we attempt to take advantage of the concept of nearest-neighbor search (used to reduce the lookup time of nearest neighbor for the candidate test case) to speed up this process, avoiding redundant distance calculations. Besides, NNDC-ART also makes use of the concept of divide-and-conquer to evenly distribute test cases over the input domain, which attempts to alleviate the boundary effect problem.

To evaluate the proposed technique, we conducted a series of simulations and empirical studies on 19 C++ or Java programs. Meanwhile, we compared our NNDC-ART approach against two enhancements of FSCS-ART based on the partitioning strategy, i.e., *ART with Divide-and-Conquer* (ART-DC) [25] and *ART by Bisection and Localization* (ART-B-Loc) [23], and further analyzed the differences among them. In summary, the main contributions of this paper are summarized as:

- We propose an alternative ART approach to enhance FSCS-ART based on the concepts of nearest-neighbor and divide-and-conquer, namely *Nearest-Neighbor Divide-and-Conquer based ART* (NNDC-ART).
- We report on simulations and empirical studies investigating NNDC-ART, from the perspectives of: testing effectiveness (i.e., test case distribution and failure detection capability), and testing efficiency (i.e., test case generation time).
- Compared with FSCS-ART, our proposed approach significantly reduces computational overhead while maintaining comparable failure detection effectiveness in low-dimensional input domains; but achieving better failure detection performances in high-dimensional input domains. In addition, we further compare NNDC-ART with ART-DC and ART-B-Loc, and find that our approach can achieve similar or even better testing performances in some scenarios.
- Our proposed approach obviously alleviates the boundary effect problem of FSCS-ART to some extent, because it provides highly better test case distribution than FSCS-ART.

The paper is organized as follows: Section 2 briefly introduces some background information; while Section 3 describes the details of the proposed approach. Section 4 provides the setup of simulations and empirical studies; while Section 5 presents the experimental results. Section 6 lists some related work; while Section 7 concludes the paper, and discusses some potential future work.

[1] The failure region is a set of all *failure-causing inputs*, each of which can trigger software failures.



(a) Strip pattern (b) Block pattern (c) Point pattern

Fig. 1. An illustrative example of three failure patterns in 2-dimensional input domain.

2. Background

In this section, we briefly present some background information about failure patterns, and *Fixed-Sized-Candidate-Set ART* (FSCS-ART).

2.1. Failure patterns

For a faulty program, a *failure-causing input* refers to a program input triggering software to exhibit failure behaviours; while the *failure region* is a set of all failure-causing inputs. Generally speaking, there are two basic features to describe the failure region: *failure pattern* and *failure rate*. The failure pattern refers to the shape of the failure region together with its distribution over the input domain; while the failure rate, denoted by θ in this paper, refers to the ratio between the size of the failure region and that of the input domain.

Many studies have individually investigated failure patterns, and observed that failure-causing inputs normally tend to be clustered into contiguous failure regions over the input domain [10–14]. Chan et al. [26] classified failure patterns into three major types: *strip pattern*, *block pattern*, and *point pattern*. For the strip pattern, its main characteristic is that failure-causing inputs are clustered into the shape of a narrow strip; while the main characteristic of the block pattern is that failure-causing inputs are concentrated in either one or a few contiguous regions. However, for the point pattern, its main characteristic is that either failure-causing inputs are standalone points, or they form regions with a very small size which are scattered over the input domain. Chan et al. [26] also claimed that strip and block patterns are more commonly encountered than point patterns in practical programs. Fig. 1 provides an example of these three failure patterns in a 2-dimensional input domain, where the bounding box represents the boundary of the input domain; and the black block, strip, and dots represent the failure-causing inputs.

2.2. Fixed-sized-candidate-set ART

Adaptive Random Testing (ART) [9] attempts to achieve an even spread of random test cases over the input domain. As summarized in [9], there are various algorithms to implement ART, according to different criteria.

The first implementation of ART is called *Fixed-Sized-Candidate-Set ART* (FSCS-ART) [15], which is also the most well-known version. It takes advantage of two sets of test cases, the *executed set* denoted by E and the *candidate set* denoted by C. The former is to store all executed test cases which have already been executed without revealing any failure; while the latter is to store the k candidate test cases which are randomly generated from the input domain (according to uniform distribution), where k is a constant throughout the testing process.[2] A candidate c will be selected from C as the next test case such that it has the longest distance to its nearest neighbor in E, i.e.,

$$\forall c' \in C,\ \min_{e \in E} dist(c, e) \geq \min_{e \in E} dist(c', e), \tag{1}$$

where $dist(c, e)$ refers to the Euclidean distance between two inputs c and e. Fig. 2 gives an illustrative example of the FSCS-ART process in a 2-dimensional input domain. As shown in Fig. 2(a), E contains three previously executed test cases, t_1, t_2, and t_3, while C contains two randomly generated test candidates, c_1 and c_2. To select the next test case from C, each candidate is required to calculate the distance against its neighbor in E. As shown in Fig. 2(b), the nearest neighbor of c_1 is t_2; while the nearest neighbor of c_2 is t_1. At last, the candidate with the longest distance is selected to be the next test case, i.e., c_2 becomes the fourth test case t_4 (Fig. 2(c)).

[2] As recommended by Chen et al. [15], the default value of k is assigned by 10.

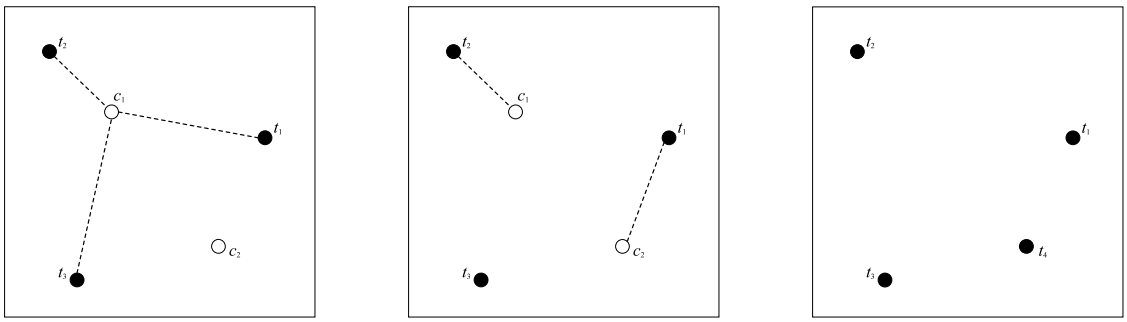


Fig. 2. An illustrative example of FSCS-ART in a two-dimensional input domain.

Many previous investigations have shown that FSCS-ART is more effective than RT according to different evaluation criteria such as fault detection capability [15–18], test case distribution [19], and code coverage [20]. However, FSCS-ART suffers from high computational overhead, because its time complexity order is $O(N^2)$ when generating N test cases [15]. In effect, a high computational overhead of FSCS-ART is required to identify the nearest neighbor for each candidate. As a result, to reduce the computational overhead, the point is to search the nearest neighbors for candidates as soon as possible, known as the *Nearest-Neighbor Search* (NNS) [27]. In addition, FSCS-ART also suffers from the boundary effect problem, as its generated test cases are likely to spread around the boundary of the input domain [21].

3. Adaptive random testing based on nearest-neighbor and divide-and-conquer

In this section, we present a new alternative ART approach, namely *Nearest-Neighbor Divide-and-Conquer based ART* (NNDC-ART), and then provide an analysis of the space and time complexity for NNDC-ART.

NNDC-ART makes use of the concept of "*divide-and-conquer*" for dividing a large problem into smaller sub-problems. Specifically, the proposed approach uses bisectional partitioning to partition the input domain into smaller subdomains. Then FSCS-ART is applied to each of the subdomains, i.e., a test case is generated from each sub-domain. Once certain conditions are satisfied, such as the computational operation is too expensive for applying the ART algorithm again, the sub-problems are decomposed and this progress is repeated recursively until either a first failure is detected or the time limit is reached. In addition, NNDC-ART uses the concept of "*nearest-neighbor*" to reduce the time of searching the nearest neighbor for each random candidate.

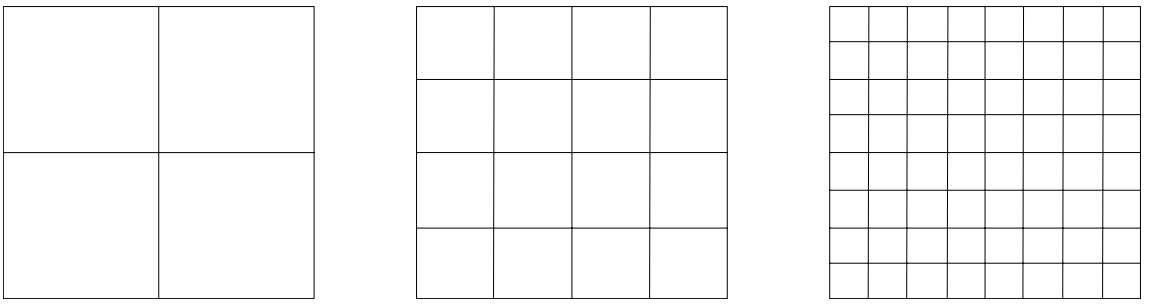
3.1. Partitioning schema

Consider a d-dimensional input domain $\mathcal{D}$, the partitioning schema first bisects each dimension of $\mathcal{D}$, leading to 2^d equal-sized subdomains $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_{2^d}$ that satisfy the following properties: 1) $\mathcal{D} = \bigcup_{i=1}^{2^d} \mathcal{D}_i$; 2) $\mathcal{D}_i \bigcap \mathcal{D}_j = \emptyset$, where $i, j = 1, 2, \cdots, 2^d$ and $i \neq j$; and 3) $|\mathcal{D}_1| = |\mathcal{D}_2| = \cdots = |\mathcal{D}_{2^d}|$, where $|\cdot|$ denotes the size of a domain. Once the condition that makes subdomain repartition is reached (for example, the number of test cases in each subdomain reaches the predefined threshold value), the next partitioning will be conducted by applying the above bisectional division process on each subdomain recursively. In this paper, the condition of dynamic adjustment is that every subdomain contains one test case. To further describe the number of subdomains, a *depth* value δ is to represent how many times such partitioning schema has been applied. For a given value of δ, there could be $2^{\delta * d}$ subdomains for the d-dimensional input domain.

Fig. 3 illustrates an example of the partitioning schema in a 2-dimensional input domain. As shown in Fig. 3(a), the input domain $\mathcal{D}$ is first divided into $2^{1*2} = 4$ equal-sized subdomains, where δ=1 and d=2. As shown in Fig. 3(b), the next partitioning iteration is conducted to divide $\mathcal{D}$ into $2^{2*2} = 16$ subdomains with the same size, where $\delta = 2$. Similarly, as shown in Fig. 3(c), $\mathcal{D}$ is divided into $2^{3*2} = 64$ subdomains through the third partitioning (i.e., $\delta = 3$).

3.2. Test case generation for each subdomain

In each subdomain, only one test case is required to be generated. Here, we applied the principle of FSCS-ART [15] to guide test case generation, which contains three similar steps: 1) randomly selecting k candidate test inputs from the subdomain (namely, the *source subdomain*); 2) identifying the nearest neighbor from previously executed test cases for each candidate; and 3) choosing an element from candidates as the next test case such that it has the longest distance against its nearest neighbor.



(a) First partitioning (b) Second partitioning (c) Third partitioning

Fig. 3. The partitioning schema in a 2-dimensional input domain.

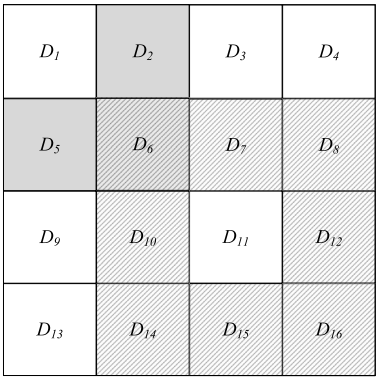


Fig. 4. An example of adjacent subdomains in a 2-dimensional input domain.

3.2.1. Random candidate construction

Similar to FSCS-ART, here k test inputs are randomly selected from the source subdomain $\mathcal{D}_i$ $(1 \leq i \leq 2^{\delta * d})$ according to uniform distribution, to construct a candidate set $C_i = \{tc_1, tc_2, \cdots, tc_k\}$. Following previous study [15], the default value of k is set as 10.

3.2.2. Adjacent subdomain identification

The original FSCS-ART calculates the distances of a candidate against all previously executed test cases, which may bring high computational overhead, especially when the number of test cases is large. In this paper, we take advantage of the positions of subdomains to highly reduce the number of executed test cases that are required to be further calculated, thereby saving computational time.

For each source subdomain $\mathcal{D}_i$ $(1 \leq i \leq 2^{\delta * d})$, there are some subdomains that are adjacent to $\mathcal{D}_i$ (these subdomains are named *adjacent subdomains*). Take a 2-dimensional input domain $\mathcal{D}$ for example, as shown in Fig. 4, the input domain $\mathcal{D}$ is divided into 16 subdomains $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_{16}$. The source subdomain $\mathcal{D}_1$ has only 3 adjacent subdomains, i.e., $\mathcal{D}_2$, $\mathcal{D}_5$, and $\mathcal{D}_6$; while another source subdomain $\mathcal{D}_{11}$ has 8 adjacent subdomains, i.e., $\mathcal{D}_6$, $\mathcal{D}_7$, $\mathcal{D}_8$, $\mathcal{D}_{10}$, $\mathcal{D}_{12}$, $\mathcal{D}_{14}$, $\mathcal{D}_{15}$, and $\mathcal{D}_{16}$. Intuitively speaking, a source subdomain has at most $(3^d - 1)$ adjacent subdomains.

3.2.3. Nearest-neighbor calculation

After adjacent subdomains are identified, since each subdomain allows only one test case inside, i.e., it is guaranteed by the construction that there is an adjacent subdomain with an executed test case, so there are at most $(3^d - 1)$ previously executed test cases that are required to be further calculated against candidates. In other words, it is not necessary to calculate the distances of candidates against all executed test cases. Obviously, it may happen that there are no executed test cases in the adjacent regions of the source subdomain, which means the input domain has a large space that is not tested. In this case, according to the principle of *discrepancy* [19], NNDC-ART would randomly select one test case from the candidates. Similar to FSCS-ART, the Euclidean distance is applied to the calculation between candidates (in the source subdomain) and executed test cases (in the adjacent subdomains). Then, the candidate will be chosen as the next test case that has the longest distance against executed test cases from the adjacent subdomains.

3.3. Algorithm

Algorithm 1 describes the detailed information about NNDC-ART. More specifically, the first step is to adopt the partitioning schema (as discussed in Section 3.1), in order to divide the input domain $\mathcal{D}$ into $m = 2^{\delta * d}$ equal-sized subdomains

Algorithm 1 NNDC-ART.

Set $E \leftarrow \emptyset$ /* the executed set to store executed test cases in the d-dimensional input domain $\mathcal{D}$ */
Set $\delta \leftarrow 0$ /* the depth of the partitioning schema */
while (*Stopping criteria not satisfied*)
  Bisectionally partition $\mathcal{D}$ into m subdomains $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$, where $m = 2^{d*\delta}$;
  Set *s_list* $\leftarrow \{\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m\}$; /* a list to store subdomains */
  Assign all executed test cases in E into m subsets $E_1, E_2, \cdots, E_m$ in terms of the distribution of $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$, and then set *e_list* $\leftarrow \{E_1, E_2, \cdots, E_m\}$; /* a list to store subsets of executed test cases distributed within the corresponding subdomains */
  while ($|E| \neq m$) **or** (*Stopping criteria not satisfied*)
    Designate a subdomain $\mathcal{D}_i$ without any test cases from *s_list* as the source subdomain; /* $\mathcal{D}_i$ such that $E_i = \emptyset$, where $i = 1, 2, \cdots, m$ */
    Identify all adjacent subdomains of $\mathcal{D}_i$ from *s_list*, and then insert them into *a_list*; /* a list to store adjacent subdomains */
    Randomly generate k test inputs $c_1, c_2, \ldots, c_k$ from $\mathcal{D}_i$, according to the uniform distribution, and then set $C_i \leftarrow \{c_1, c_2, \ldots, c_k\}$;
    if *a_list* $= \emptyset$
      Randomly select a test case c_b from C_i
    else
      for (each candidate $c_j \in C_i$, where $j = 1, 2, \cdots, k$)
        for (each element $\mathcal{D}_r \in$ *a_list*, where $r \in [1, m]$)
          Calculate the distance between c_j and e, where $e \in E_r$;
        end_for
        Assign the smallest distance for c_j denoted as d_j;
      end_for
      Find $c_b \in C_i$ such that $d_b \geq d_j, \forall j \in [1, k]$;
    end_if
    Set $tc \leftarrow c_b$, and add tc into E_i and E, and execute tc.
  end_while
  Set $\delta \leftarrow \delta + 1$;
end_while

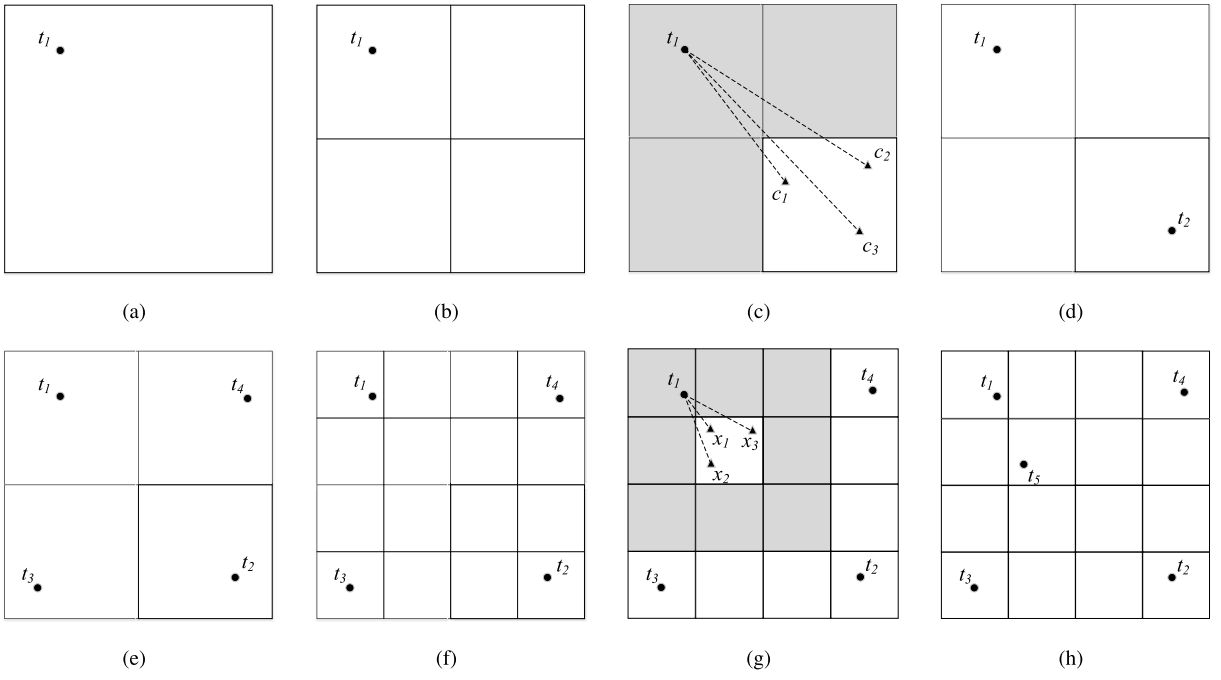


Fig. 5. An illustrative example of NNDC-ART in a 2-dimensional input domain.

$\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$. NNDC-ART then assigns previously executed test cases (that are stored in the executed set E) into their relevant subdomains, to obtain $E_1, E_2, \cdots, E_m$ that denote the sets of executed test cases in $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$, respectively. Regarding test case generation, it is necessary to designate a subdomain from $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$ as the source domain for generating the next test case. Since only one test case is allowed to be selected from each subdomain, the subdomains without any test case could be designated as source subdomains. When there is more than one source subdomain, a random one is selected. After that, the next test case will be generated from the selected source domain, as discussed in Section 3.2. When the number of total test cases reaches the number of all subdomains, i.e., each subdomain contains one test case, NNDC-ART will take the next round of the partitioning schema. This process will be repeated until certain stopping criteria are satisfied (such as "a certain number of test cases have been generated", "a software failure has been identified", and "testing resources have been exhausted").

Fig. 5 shows an example to illustrate the detailed process of NNDC-ART in the 2-dimensional input domain. As shown in Fig. 5(a), the first test case t_1 is randomly generated from the input domain, according to uniform distribution, and then the partition schema is applied to divide the input domain (as shown in Fig. 5(b)). A subdomain with no test cases is designated as the source subdomain (the right bottom subdomain in Fig. 5(c)), and then three candidates c_1, c_2, and c_3 are randomly selected from the source subdomain. After calculating the distances between candidates and elements in the adjacent subdomains (the shaded subdomains), the candidate c_3 becomes the next test case t_2 (as shown in Fig. 5(d)). Similarly, another two subdomains without any test cases will generate the corresponding test cases t_3 and t_4, respectively (as shown in Fig. 5(e)). Because the partitioning condition is reached, NNDC-ART applied the next partitioning schema to the input domain, resulting in 16 subdomains (as shown in Fig. 5(f)). Similar to Fig. 5(c), a source subdomain randomly generates three candidates x_1, x_2, and x_3, and only one previously generated test case (i.e., t_1) is required to be calculated against each candidate (because other test cases t_2, t_3, and t_4 are not located in the adjacent subdomains in this case). After calculation, x_2 becomes the fifth test case, i.e., t_5, as shown in Fig. 5(h).

3.4. Complexity analysis

In this section, we briefly analyze the space and time complexity of NNDC-ART from a formal mathematical perspective.

3.4.1. Space complexity

As shown in Algorithm 1, NNDC-ART needs memory space to store N executed test cases and all $2^{d*\delta}$ subdomains. As we know, since each subdomain contains at most one test case, it can be concluded that $2^{d*(\delta-1)} \leq N \leq 2^{d*\delta}$, indicating that $2^{d*\delta}$ is at most in $O(N)$. As a consequence, the order of space complexity for NNDC-ART is $O(N)$.

3.4.2. Time complexity

The time complexity of the NNDC-ART algorithm mainly depends on the process of generating new test cases.

For d-dimensional input domain, NNDC-ART needs to check at most $(3^d - 1)$ adjacent subdomains, whether they contain the executed test cases or not. Once the number of dimensions is confirmed, the maximum number of neighboring regions are also fixed. For FSCS-ART, the selection of test cases involves the process that is distance computation between the candidate test case and executed test cases. However, the size of executed test cases will be continuously expanded with the progress of testing, which results in high computational overhead. Since this process for NNDC-ART is limited to the constant number of executed test cases (i.e., the limited number of adjacent subdomains), the generation of test cases always requires constant time. Therefore, it can be concluded that the order of time complexity of NNDC-ART is $O(N)$ to generate N test cases.

It should be noted that NNDC-ART involves the process of partitioning region and assigning executed test cases to the corresponding subdomains. In NNDC-ART, when each subdomain region has one executed test case, the input domain will be divided into $2^{d*\delta}$ new subdomains by bisecting each dimension, therefore, the process of the partitioning takes time in $O(2^{d*\delta})$. Except for the time overhead of the partitioning, only one executed test case in the previous region will be reallocated into the corresponding new $2^{d*\delta}$ subdomains, which also requires the time in $O(2^{d*\delta})$. Nevertheless, the region partition does not occur in the generation of test cases every time, only when $N = 2^{\delta*d}$. Therefore, compared with the time overhead required to generate the test case, the time cost for this process will not affect the time complexity of NNDC-ART. To conclude, NNDC-ART generates test cases in linear time, i.e., $O(N)$.

4. Experimental studies

To validate the effectiveness of the NNDC-ART approach, we conducted a series of simulations and empirical studies with different research objects. The specific experimental designs and settings are introduced in this section.

4.1. Research objects

Except for FSCS-ART and NNDC-ART algorithm, we also use another two enhancements of FSCS-ART based on the partitioning strategy as the research objects, i.e., ART-B-Loc [23] and ART-DC [25]. A brief description of these two algorithms is described as follows.

ART-B-Loc has been proposed for improving the effectiveness of ART by Bisection, which combines ART by bisection and the principle of localization [23]. Although ART-B-Loc also applies dynamic partitioning and divide-and-conquer, there are some differences between it and NNDC-ART: 1) Different numbers of neighbor subdomains to generate potential test cases. Specifically, for each source subdomain $\mathcal{D}_i$, the adjacent subdomains is $2 * d$ for ART-B-Loc, while NNDC-ART has at most $(3^d - 1)$ adjacent subdomains. Intuitively speaking, the higher the number of adjacent regions to check, the more accurate the minimum distance between the candidate test cases and the executed test cases. Therefore, NNDC-ART may keep the effectiveness of ART as much as possible. 2) Different ways to construct candidate test cases. When generating the candidate test cases, ART-B-Loc randomly selects one empty subdomain to generate a candidate test case, which repeats k times. However, NNDC-ART selects one empty region to create k candidates. Obviously, in the process of nearest-neighbor calculation, NNDC-ART only needs to consider the adjacent subdomains of an empty region. In contrast, ART-B-Loc needs

to identify the adjacent subdomains for each region where candidate test cases are generated, thus incurring more time overhead.

ART-DC is an efficient ART algorithm by applying the concept of divide-and-conquer [25]. It first uses FSCS-ART to generate test cases from the whole input domain. Once the number of test cases reaches a predefined threshold (denoted by λ), the input domain is divided into some equal-sized subdomains by bisectionally dividing each dimension, and then test cases are generated from these subdomains independently by using FSCS-ART. Especially when $\lambda = 1$, ART-DC will randomly generate a test case within each empty subdomain, which can be considered as a special case of ART-B. Although NNDC-ART also generates a test case in each empty region, it utilizes the information of the executed test cases located in the adjacent regions of the source subdomain. Therefore, NNDC-ART may achieve a better test case distribution by calculating distances against a limited number of adjacent subdomains.

4.2. Research questions

We propose NNDC-ART to further reduce the computation overhead of FSCS-ART, and also to alleviate the boundary effect problem. Intuitively speaking, it is required to not only examine the test case generation time of the new proposed approach, but also check its test case distribution. As we know, while decreasing the computation overhead, it is expected that NNDC-ART would maintain the comparable failure-detection effectiveness to FSCS-ART, to deliver a high cost-effectiveness in testing. Therefore, it is also needed to evaluate NNDC-ART at revealing software failures under different scenarios. In addition, it is also necessary to investigate the comparisons of NNDC-ART against some enhancements of FSCS-ART based on the partitioning strategy, such as ART-B-Loc and ART-DC. The experimental studies help us answer the following three research questions:

RQ1 *Does NNDC-ART alleviate the boundary effect problem of FSCS-ART?*
RQ2 *Compared with FSCS-ART and its enhancements such as ART-B-Loc and ART-DC, how effective is NNDC-ART at identifying software failures?*
RQ3 *How is the computation overhead of NNDC-ART compared with other ART algorithms?*

4.3. Evaluation metrics

In this section, we describe some evaluation metrics (including effectiveness and efficiency metrics) to evaluate the proposed approach against different ART algorithms.

4.3.1. Effectiveness metrics

To measure the capability of test case distribution, the *discrepancy* has been popularly used, which measures the equidistribution of sample inputs [19], with lower values indicating better distributions. The discrepancy of a test set T is defined as:

$$M_{Discrepancy}(T) = \max_{1 \le i \le n} \left| \frac{|T_i|}{|T|} - \frac{|\mathcal{D}_i|}{|\mathcal{D}|} \right|, \tag{2}$$

where $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_n$ are n randomly defined subdomains of the input domain $\mathcal{D}$; and $T_1, T_2, \cdots T_n$ are the corresponding subsets of T, satisfying that all elements in T_i $(i = 1, 2, \cdots, n)$ are located in $\mathcal{D}_i$. Following previous studies [19], n was set as 1000 in this paper. Discrepancy checks whether or not the number of test cases in a subdomain is proportionate to the relative size of the subdomain area, which indicates that smaller subdomains should have relatively fewer, and larger ones should have more test cases.

The *F-measure* [16] is a metric to evaluate and compare the failure-detection effectiveness of ART algorithms, which has been widely adopted by many studies on ART [9]. The *F-measure* refers to the number of expected test cases required to detect the first software failure. In this paper, F_{RT} and F_{ART} denote the F-measure of RT and ART respectively. For the case of the random selection of test cases with replacement, the F_{RT} is equal to $1/\theta$. For ease of description, the *F-ratio* is used to denote the ratio of F_{ART} to F_{RT}, which measures the F-measure improvement of ART over RT. The smaller the *F-ratio*, the better performance of the ART algorithm against RT.

4.3.2. Efficiency metrics

The execution time is an intuitive metric for evaluating testing efficiency of the testing techniques to satisfy certain stopping criteria. As reported in the next section, we collected the average execution time of different ART algorithms to generate the same number of test cases in the simulations; while in the empirical studies, we measured the average execution time to identify the first failure for each real-life program (i.e., F_{time} [28]).

4.4. Simulations and empirical studies

To evaluate NNDC-ART against FSCS-ART and its enhancements based on the partitioning strategy, we conducted a series of simulations and empirical studies. The simulations are to simulate software program faults; while empirical studies are to adopt seeded-faults in the programs by mutating source code.

4.4.1. Design of simulations

In our simulations, a d-dimensional unit hypercube is often used to simulate the input domain $\mathcal{D}$. Without loss of generality, we set $\mathcal{D} = \{(x_1, x_2, \cdots, x_d) | 0 \leq x_1, x_2, \cdots, x_d < 1.0\}$. For the dimension d of the input domain, it was set as 1, 2, 3, 4, 5, and 8 in this study.

To address RQ1, it is required to construct the set of test cases (i.e., E). To provide sufficient comparisons, we selected different sizes for E, i.e., the size of E was set as from 100 to 10,000.

To address RQ2, we simulated failure regions randomly placed inside $\mathcal{D}$. As we know, the size (described by the *failure rate* θ) and shape (described by the *failure pattern*) of failure regions are fixed after coding but unknown before testing. In the simulations, therefore, both failure rate and failure pattern were defined in advance, and the failure regions were randomly placed in the input domain. During the testing process, when a test case is selected inside a failure region, a failure is said to be triggered. As mentioned in Section 2.1, there are three main categories of failure patterns, i.e. *block*, *strip*, and *point patterns*. In the simulations, we adopted these three failure patterns to simulate failure regions within the input domain. For the block pattern, the failure-causing inputs are clustered into a single region with equal lengths of side (such as a square/cube in 2/3-dimensional spaces), then a single failure region was randomly placed inside the input domain. For the strip pattern, a strip region is created at any angle by randomly choosing two points on adjacent borders of the domain. To simulate the point pattern, 25 equal-sized and non-overlapping failure regions are randomly placed in the input domain according to the predetermined failure rate. Regarding the failure rate θ, it was set as 0.01, 0.0075, 0.005, 0.0025, 0.001, 0.00075, 0.0005, 0.00025, and 0.0001, respectively.

To address RQ3, we conducted simulations to evaluate the execution time of test case generation for NNDC-ART, FSCS-ART, ART-B-Loc, and ART-DC, respectively. In the simulations, we adopted each ART algorithm to generate totally 20,000 test cases in the d-dimensional input domain. The execution time was collected to generated N test cases, where N ranges from 500 to 20,000 with the increment 500.

4.4.2. Design of empirical studies

Simulations can simulate the performance of the algorithm under different conditions, however, they may suffer from some limitations. For example, there are a lot of influencing factors, and the types of failure are often not single in real-life programs. Previous ART studies [15,18,29,30] have used 12 fault-seeded programs to evaluate the effectiveness of ART algorithms in real-life programs. These published programs were taken from ACM's collected algorithms [31] and the Numerical Recipes [32]. In this study, we also adopted these 12 object programs. In addition, we selected another 7 real-life programs with the high-dimensional input domain. The programs `calDay`, `complex`, and `line` were acquired from Ferrer et al. [33]. The position relationships between points and a triangle, a point and a line segment, and two line segments are determined through the programs `pntTrianglePos`, `pntLinePos`, and `twoLinesPos`, respectively. The triangle is classified into one of three types: acute, right, and obtuse through the `triangle` program. These four programs including `pntTrianglePos`, `pntLinePos`, `twoLinesPos` and `triangle` are adopted by KDFC-ART [34] to investigate the failure detection ability of ART algorithms. Overall, we used 19 real-life programs to investigate the fault detection performances of NNDC-ART as compared to FSCS-ART, ART-B-Loc, and ART-DC.

Some faults were seeded into each program using the following six common mutation operators [35]: *relational operator replacement* (ROR), *arithmetic operator replacement* (AOR), *constant replacement* (CR), *statement deletion* (SDL), and *scalar variable replacement* (SVR). The dimension of the input domain ranges from 1 to 8, while the failure rate ranges from about 0.0001 to 0.002. Table 1 summarizes the detailed information of these 19 real-life programs.

4.5. Experiment environment

The simulations were conducted on a machine serving Windows 10 Home, equipped with an Intel(R) Core(TM) i5-6200U CPU (2.30 GHz, 4 core) with 16 GB of RAM. The used IDEs are Microsoft Visual Studio 2013 (Visual C++) and Eclipse (Neon Release (4.6.0)).

4.6. Data collection and statistical analysis

For evaluating the F-measure of the experiments, test cases are generated (using ART algorithms) and executed until a failure is detected. For simulations, when a point is picked up from the simulated failure region, it can be said that a failure is detected; while for empirical studies, the original programs are used as the test oracles for their mutants. To provide sufficient comparisons, we run each experiment 3000 times, and report the average result.

In this paper, we computed both *p-value* (probability value) and *effect size* to examine the significance of the performance differences between NNDC-ART and other ART algorithms. The unpaired two-tailed Mann-Whitney-Wilcoxon test [36] is

Table 1
Subject programs for empirical study.

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

AOR: arithmetic operator replacement, ROR: relational operator replacement, SVR: scalar variable replacement, CR: constant replacement, and SDL: statement deletion.

used to validate whether there are statistically significant differences among the investigated ART algorithms. The p-value is less than 0.05, indicating a significant difference between the compared two methods. Meanwhile, the effect size [36] is used to show the probability that one method is superior to the other, which will be complemented through the non-parametric Vargha and Delaney effect size [37]. More specifically, for two methods X and Y, the value of effect size represents the probability that X is better than Y. If the value of effect size is equal to 0.5, these two algorithms are equivalent, while the value of effect size is greater than 0.5, indicting that X is better than Y.

5. Experimental results

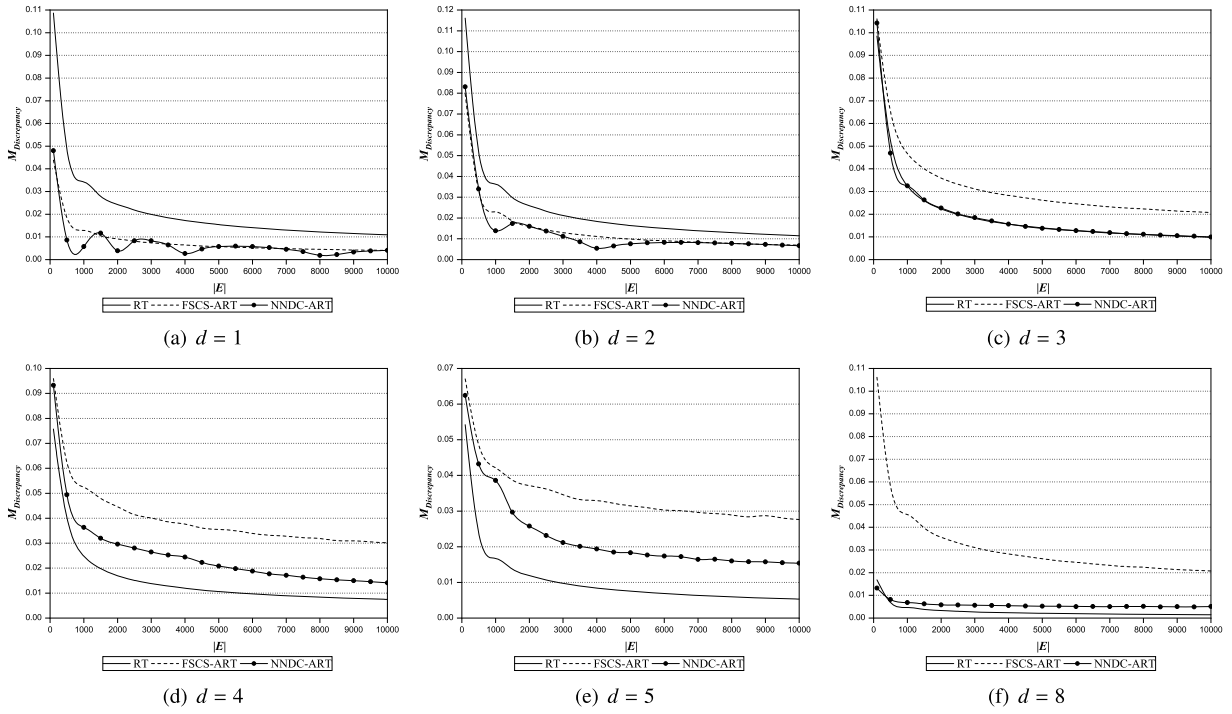
In this section, we provide the experimental results and discussions for evaluating NNDC-ART against other ART algorithms (including FSCS-ART, ART-B-Loc, and ART-DC), to answer the above three research questions.

5.1. Answers to RQ1: test case distribution

Fig. 6 shows the simulation results of test case distribution measured by the discrepancy as defined in Equation (2). From the data, we can have the following observations:

(1) Compared with RT, when d is equal to 1 or 2, both NNDC-ART and FSCS-ART achieve better test case distribution. When d is equal to 3, NNDC-ART performs slightly better or very similar to RT, however, FSCS-ART is worse than RT. For other cases with $d \geq 4$, both NNDC-ART and FSCS-ART perform worse than RT.
(2) Compared with FSCS-ART, when d is smaller than 3, NNDC-ART has slightly better or similar discrepancy. However, when $d \geq 3$, NNDC-ART performs much better than FSCS-ART, regardless of the sizes of test case sets. In addition, with the increase of the number of test cases, the discrepancy differences between NNDC-ART and FSCS-ART become larger.

The above observations are easy to be explained. On the one hand, as described in Section 3, the divide-and-conquer strategy is adopted to distribute test cases over the input domain, and to guarantee that each subdomain contains only one test case. In addition, the nearest-neighbor strategy is used to guarantee that each test case in each subdomain achieves a certain distance against its neighbors. Therefore, NNDC-ART can generate more evenly distributed test cases than FSCS-ART over the input domain. On the other hand, similar to other ART algorithms [9], NNDC-ART also suffers from the "*curse of dimensionality*" [38] (resulting in its worse performances than RT), when the dimension becomes large.



(a) $d = 1$ (b) $d = 2$ (c) $d = 3$

(d) $d = 4$ (e) $d = 5$ (f) $d = 8$

Fig. 6. $M_{Discrepancy}$ of NNDC-ART.

Summary of Answers to RQ1: *NNDC-ART achieves similar or slightly better test case distribution than FSCS-ART for low-dimensional input domains; however, with the increase of the dimension, NNDC-ART significantly outperforms FSCS-ART. In other words, it can be concluded that NNDC-ART can effectively alleviate the boundary effect problem to some extent.*

5.2. Answers to RQ2: fault detection effectiveness

5.2.1. Results of simulations

Tables 2 to 4 present the F-ratio comparisons between NNDC-ART and other ART algorithms based on simulations with the block, strip, and point patterns. In addition, the p-values and effect size values were collected between NNDC-ART and other ART algorithms. According to the experimental data from these tables, we can observe the followings:

(1) Similar to FSCS-ART, when considering a given dimension d, NNDC-ART generally has lower F-ratio values with the decrease of the failure rate θ for the block and point patterns. In other words, the smaller the failure rate is, the better performance of NNDC-ART is (as compared to RT).

(2) For the block patterns (as shown Table 2): When considering a failure rate θ, the F-ratio values of NNDC-ART are higher, with the increase of the dimension d. In other words, the higher the dimension is, the worse performance of NNDC-ART is (it can be seen that FSCS-ART also has the similar observation). In detail, when $1 \leq d \leq 4$, NNDC-ART performs much better than RT in most cases. However, when d becomes higher, NNDC-ART is worse than RT overall. Compared with FSCS-ART, NNDC-ART achieves comparable F-ratio values, when d is equal to or less than 4. However, when d becomes higher, NNDC-ART is better than FSCS-ART overall. The statistical analysis generally confirms the above observations. For example, when d ranges from 1 to 4, most of p-values are higher than 0.05, and nearly all effect size values are approximately equal to 0.50. In addition, when $d = 5$, the majority of p-values are less than 0.05, indicating that the F-ratio differences between NNDC-ART and FSCS-ART are highly significant; while the effect size values range from 0.50 to 0.53. Moreover, when d is equal to 8, all p-values are much less than 0.05 (except the case of $\theta = 0.01$); and most of the effect size values are larger than 0.56.

Compared with ART-B-Loc and ART-DC, NNDC is more susceptible to dimensions. When $1 \leq d \leq 4$, most of effect size values are larger than 0.5, which means that NNDC-ART has a slightly better failure detection capability. When $d \geq 5$, more effect size values are less than 0.5, which means that NNDC-ART starts to be worse than ART-B-Loc and ART-DC. From the comparison results between NNDC-ART and other algorithms, we can observe that no algorithm always achieves good testing results in general. Specifically, when the input domain dimension is low, that is, $1 \leq d \leq 3$, NNDC-ART, ART-B-Loc and ART-DC ($\lambda = 50$) perform similarly in most cases. However, NNDC-ART performs much better than ART-DC ($\lambda = 1$). In addition, when $d = 4$, NNDC-ART has similar F-ratio results to ART-B-Loc, while achieving similar or worse performances to ART-DC ($\lambda = 1$). However, NNDC-ART achieves much better performances than ART-DC ($\lambda = 50$), regardless of θ. When

Table 2
F-ratio comparisons between NNDC-ART and other ART algorithms for simulations with the *block patterns*.

Dim. (d)	Failure Rate (θ)	F-ratio					Statistical Analysis							
		FSCS-ART	ART-B-Loc	ART-DC ($\lambda=1$)	ART-DC ($\lambda=50$)	NNDC-ART	*vs.* FSCS		*vs.* ART-B-Loc		*vs.* ART-DC ($\lambda=1$)		*vs.* ART-DC ($\lambda=50$)	
							p-value	effect size	*p*-value	effect size	*p*-value	effect size	*p*-value	effect size
$d=1$	0.01	0.5577	0.5729	0.6170	0.5638	0.5567	0.6500	0.5034	0.0061	0.5204	0.0006	0.5254	0.2546	0.5085
	0.0075	0.5699	0.5721	0.6225	0.5614	0.5622	0.0461	0.5149	0.0219	0.5171	0.0000	0.5312	0.5166	0.5048
	0.005	0.5660	0.5655	0.6223	0.5638	0.5802	0.4427	0.4943	0.6378	0.4965	0.0912	0.5126	0.2532	0.4915
	0.0025	0.5570	0.5547	0.6242	0.5611	0.5698	0.0681	0.4864	0.3379	0.4929	0.0013	0.5240	0.2972	0.4922
	0.001	0.5563	0.5426	0.6191	0.5534	0.5524	0.5312	0.5047	0.5431	0.5045	0.0000	0.5339	0.9122	0.5008
	0.00075	0.5539	0.5637	0.6297	0.5556	0.5683	0.4128	0.4939	0.7006	0.5029	0.0000	0.5341	0.5346	0.4954
	0.0005	0.5514	0.5457	0.6153	0.5656	0.5670	0.1765	0.4899	0.2591	0.4916	0.0096	0.5193	0.7842	0.5020
	0.00025	0.5480	0.5631	0.6153	0.5648	0.5686	0.0763	0.4868	0.7449	0.5024	0.0183	0.5176	0.9092	0.4991
	0.0001	0.5649	0.5485	0.6228	0.5602	0.5613	0.2753	0.5081	0.9719	0.4997	0.0000	0.5342	0.5777	0.5042
$d=2$	0.01	0.6833	0.6781	0.7546	0.6867	0.7224	0.0510	0.4855	0.0723	0.4866	0.7408	0.4975	0.0875	0.4873
	0.0075	0.6572	0.6846	0.7430	0.6836	0.6967	0.0034	0.4781	0.5125	0.4951	0.8320	0.5016	0.3478	0.4930
	0.005	0.6618	0.6774	0.7760	0.6696	0.6800	0.5840	0.4959	0.5000	0.5050	0.0007	0.5252	0.9483	0.4995
	0.0025	0.6511	0.6653	0.7639	0.6581	0.6687	0.6765	0.4969	0.5451	0.5045	0.0001	0.5292	0.9032	0.5009
	0.001	0.6500	0.6474	0.7817	0.6635	0.6608	0.4905	0.5051	0.4050	0.5062	0.0000	0.5376	0.1082	0.5120
	0.00075	0.6448	0.6547	0.7496	0.6661	0.6631	0.5806	0.5041	0.9716	0.5003	0.0051	0.5209	0.2499	0.5086
	0.0005	0.6427	0.6523	0.7369	0.6466	0.6433	0.5740	0.5042	0.1590	0.5105	0.0000	0.5308	0.6733	0.5031
	0.00025	0.6307	0.6616	0.7755	0.6604	0.6445	0.4174	0.5060	0.0152	0.5181	0.0000	0.5455	0.1225	0.5115
	0.0001	0.6337	0.6380	0.7528	0.6552	0.6410	0.5288	0.4953	0.9952	0.5000	0.0000	0.5349	0.2897	0.5079
$d=3$	0.01	0.8475	0.8265	0.8639	0.8455	0.8652	0.3391	0.5071	0.7398	0.4975	0.0017	0.4766	0.8154	0.5017
	0.0075	0.8186	0.8173	0.8676	0.8574	0.8552	0.1557	0.4894	0.1157	0.4883	0.0150	0.4819	0.9863	0.5001
	0.005	0.7865	0.8112	0.8658	0.8383	0.8125	0.0418	0.4848	0.5087	0.4951	0.9241	0.5007	0.6022	0.5039
	0.0025	0.7924	0.8024	0.8389	0.8334	0.7891	0.3354	0.5072	0.3748	0.5066	0.7913	0.4980	0.0043	0.5213
	0.001	0.7447	0.7863	0.8762	0.7926	0.7949	0.0104	0.4809	0.5387	0.4954	0.1851	0.5099	0.5351	0.4954
	0.00075	0.7550	0.7881	0.8557	0.7752	0.7630	0.5770	0.4958	0.2099	0.5093	0.0203	0.5173	0.8675	0.5012
	0.0005	0.7284	0.7752	0.8409	0.7717	0.7508	0.1954	0.4903	0.1367	0.5111	0.0056	0.5206	0.6661	0.5032
	0.00025	0.7221	0.7446	0.8691	0.7918	0.7527	0.5787	0.4959	0.9284	0.5007	0.0001	0.5297	0.0073	0.5200
	0.0001	0.7221	0.7507	0.8533	0.7707	0.7482	0.6181	0.4963	0.5000	0.5050	0.0000	0.5316	0.1344	0.5112
$d=4$	0.01	1.0498	1.0215	0.8921	1.0383	1.0131	0.1073	0.5120	0.0345	0.5158	0.0000	0.4421	0.1336	0.5112
	0.0075	1.0208	0.9697	0.9021	1.0340	0.9909	0.1606	0.5105	0.0165	0.5179	0.0000	0.4544	0.3560	0.5069
	0.005	0.9675	0.9532	0.9073	1.0151	0.9798	0.3646	0.5068	0.1065	0.5120	0.0000	0.4634	0.0122	0.5187
	0.0025	0.9473	0.8811	0.9368	0.9644	0.9543	0.5283	0.5047	0.2859	0.5080	0.0001	0.4717	0.4909	0.5051
	0.001	0.8971	0.8953	0.9026	0.9449	0.8965	0.7433	0.4976	0.7794	0.5021	0.0305	0.4839	0.0855	0.5128
	0.00075	0.8751	0.8747	0.9001	0.9718	0.8528	0.3065	0.5076	0.0614	0.5139	0.6763	0.5031	0.0000	0.5354
	0.0005	0.8860	0.9147	0.9192	0.9318	0.8413	0.0044	0.5212	0.0102	0.5191	0.5085	0.5049	0.0032	0.5219
	0.00025	0.8623	0.8530	0.9456	0.9319	0.8646	0.0404	0.5153	0.0074	0.5200	0.1086	0.5120	0.0002	0.5279
	0.0001	0.8314	0.8328	0.9213	0.8899	0.8316	0.6339	0.4964	0.0426	0.5151	0.1574	0.5105	0.0300	0.5162
$d=5$	0.01	1.2876	1.0571	0.9716	1.0390	1.2254	0.3650	0.5068	0.0000	0.4413	0.0000	0.4016	0.0000	0.4478
	0.0075	1.2772	1.1111	0.9424	1.0829	1.1896	0.0084	0.5197	0.0000	0.4621	0.0000	0.4005	0.0000	0.4636
	0.005	1.2096	1.0770	0.9469	1.1260	1.1478	0.0219	0.5171	0.0003	0.4727	0.0000	0.4242	0.0062	0.4796
	0.0025	1.1672	1.0589	0.9692	1.1471	1.0853	0.0008	0.5250	0.2252	0.4910	0.0000	0.4501	0.4926	0.5051
	0.001	1.0902	0.9436	0.9411	1.0181	1.0464	0.0543	0.5143	0.0001	0.4710	0.0000	0.4544	0.4151	0.4939
	0.00075	1.0480	0.9645	0.9576	1.0411	1.0415	0.6824	0.5031	0.0093	0.4806	0.0000	0.4548	0.3366	0.5072
	0.0005	1.0751	0.9520	0.9612	1.0149	1.0085	0.0084	0.5197	0.0169	0.4822	0.0000	0.4636	0.1098	0.5119
	0.00025	1.0279	0.9720	0.9579	1.0008	0.9535	0.0032	0.5220	0.8382	0.4985	0.0336	0.4842	0.3106	0.5076
	0.0001	0.9613	0.9538	0.9440	1.0404	0.9280	0.0488	0.5147	0.5085	0.5049	0.4698	0.4946	0.0067	0.5202
$d=8$	0.01	2.6468	1.7893	0.9884	1.3511	2.4110	0.6637	0.5032	0.0000	0.3839	0.0000	0.2125	0.0000	0.2929
	0.0075	2.5330	1.6381	0.9926	1.2363	2.1803	0.0037	0.5216	0.0000	0.3842	0.0000	0.2319	0.0000	0.2914
	0.005	2.2953	1.4589	0.9821	1.1309	1.8797	0.0000	0.5445	0.0000	0.3964	0.0000	0.2641	0.0000	0.3081
	0.0025	2.1439	1.2296	1.0090	1.0315	1.6249	0.0000	0.5723	0.0000	0.4032	0.0000	0.3263	0.0000	0.3438
	0.001	1.9537	1.2211	1.0124	1.0934	1.4534	0.0000	0.5889	0.0000	0.4363	0.0000	0.3720	0.0000	0.3975
	0.00075	1.8356	1.1672	1.0234	1.1019	1.4278	0.0000	0.5744	0.0000	0.4349	0.0000	0.3878	0.0000	0.4069
	0.0005	1.8559	1.2854	1.0139	1.1807	1.3887	0.0000	0.5841	0.0002	0.4725	0.0000	0.4017	0.0000	0.4331
	0.00025	1.7426	1.2578	0.9797	1.2785	1.3694	0.0000	0.5798	0.0002	0.4723	0.0000	0.4056	0.0002	0.4722
	0.0001	1.5904	1.2758	0.9911	1.2044	1.3211	0.0000	0.5604	0.3019	0.4923	0.0000	0.4171	0.0406	0.4847

$d=5$, the F-ratio differences between NNDC-ART and ART-B-Loc are small, but ART-DC overall performs better than NNDC-ART. Similarly, when $d=8$, both ART-B-Loc and ART-DC achieve better F-ratio results than NNDC-ART.

(3) For the strip patterns (as shown in Table 3): NNDC-ART has much smaller F-measures than RT for 1-dimensional input domain; and has slightly better or similar performances for other dimensions. In addition, NNDC-ART achieves very comparable F-ratio results to other ART algorithms, regardless of dimensions, failure rates, and failure patterns. According to the statistical analysis, except for few p-values less than 0.05, the majority of p-values are greater than 0.05, which indicates that the F-ratio differences between NNDC-ART and other ART algorithms are not significant; while all effect size values range from 0.4861 to 0.5320, indicating that there is no better one between NNDC-ART and other ART algorithms overall. When $d=1$, strip and block patterns are the same in fact. When $d>1$, however, each F-ratio value is around 1.0, indicating that each ART algorithm achieves similar fault detection capability to RT.

(4) For the point patterns (as shown in Table 4): Compared with FSCS-ART, the F-ratio performances of NNDC-ART and FSCS-ART become worse along with the increase of d, irrespective of θ. More specifically, when the dimension d is small such as $d=1$ and $d=2$, both NNDC-ART and FSCS-ART have comparable F-ratio results to RT; however, when $d>2$, RT outperforms NNDC-ART and FSCS-ART. Especially when d is larger, NNDC-ART achieves better or comparable F-ratio results than FSCS-ART. In addition, when $1\leq d\leq 2$, NNDC-ART performs similarly to ART-B-Loc and ART-DC. However, when $d\geq 3$, NNDC-ART performs worse than ART-DC ($\lambda=1$) overall; while achieving similar or slightly worse performances than ART-B-Loc and ART-DC ($\lambda=50$).

According to the statistical analysis, the p-values and effect size values overall validate these observations. More specifically, when d ranges from 1 to 5, most of p-values are larger than 0.05 between NNDC-ART and FSCS-ART, and most of the effect size values are around 0.50, which indicates that their differences are not significant. However, when d is equal

Table 3
F-ratio comparisons between NNDC-ART and other ART algorithms for simulations with the *strip patterns*.

Dim. (d)	Failure Rate (θ)	F-ratio					Statistical Analysis							
		FSCS-ART	ART-B-Loc	ART-DC ($\lambda=1$)	ART-DC ($\lambda=50$)	NNDC-ART	*vs.* FSCS		*vs.* ART-B-Loc		*vs.* ART-DC ($\lambda=1$)		*vs.* ART-DC ($\lambda=50$)	
							p-value	effect size	*p*-value	effect size	*p*-value	effect size	*p*-value	effect size
$d=1$	0.01	0.5710	0.5627	0.6213	0.5683	0.5731	0.8810	0.4989	0.9544	0.4996	0.0181	0.5176	0.9654	0.5003
	0.0075	0.5594	0.5695	0.6269	0.5655	0.5730	0.4892	0.4948	0.2942	0.5078	0.0007	0.5254	0.8912	0.4990
	0.005	0.5671	0.5633	0.6217	0.5636	0.5693	0.7244	0.4974	0.7317	0.5026	0.0052	0.5209	0.4579	0.4945
	0.0025	0.5612	0.5657	0.6157	0.5619	0.5637	0.8709	0.5012	0.1552	0.5106	0.0011	0.5244	0.9772	0.5002
	0.001	0.5605	0.5553	0.6114	0.5626	0.5584	0.7628	0.5023	0.4661	0.5054	0.0090	0.5195	0.9777	0.5002
	0.00075	0.5632	0.5701	0.5916	0.5622	0.5735	0.8853	0.4989	0.3143	0.5075	0.6254	0.5036	0.9378	0.4994
	0.0005	0.5582	0.5611	0.6143	0.5527	0.5624	0.8620	0.4987	0.2032	0.5095	0.0033	0.5219	0.5035	0.4950
	0.00025	0.5624	0.5625	0.6137	0.5614	0.5527	0.1576	0.5105	0.0141	0.5183	0.0001	0.5294	0.2353	0.5088
	0.0001	0.5506	0.5583	0.6267	0.5634	0.5691	0.2531	0.4915	0.5480	0.5045	0.0000	0.5320	0.4898	0.5051
$d=2$	0.01	0.9271	0.9458	0.9250	0.9343	0.9196	0.2579	0.5084	0.1302	0.5113	0.9989	0.5000	0.6132	0.5038
	0.0075	0.9337	0.9475	0.9299	0.9592	0.9490	0.6423	0.4965	0.9838	0.4998	0.0781	0.4869	0.8737	0.4988
	0.005	0.9532	0.9735	0.9694	0.9210	0.9454	0.4148	0.5061	0.2418	0.5087	0.3095	0.5076	0.4208	0.4940
	0.0025	0.9553	0.9787	0.9617	0.9612	0.9485	0.9118	0.4992	0.1148	0.5118	0.6689	0.5032	0.5445	0.5045
	0.001	0.9898	0.9413	0.9568	0.9711	0.9361	0.0237	0.5169	0.3601	0.5068	0.1088	0.5120	0.0288	0.5163
	0.00075	0.9754	0.9796	0.9713	0.9830	0.9765	0.6538	0.5033	0.3588	0.5068	0.7782	0.5021	0.4593	0.5055
	0.0005	0.9728	0.9974	0.9370	0.9734	0.9722	0.9089	0.5009	0.9352	0.4994	0.2589	0.4916	0.7734	0.4979
	0.00025	0.9752	0.9794	0.9942	0.9836	0.9524	0.6076	0.5038	0.8736	0.5012	0.0675	0.5136	0.3049	0.5076
	0.0001	0.9729	0.9861	1.0079	0.9838	0.9831	0.4096	0.4939	0.4732	0.4947	0.8116	0.4982	0.7373	0.4975
$d=3$	0.01	0.9927	0.9648	0.9997	0.9555	0.9668	0.2992	0.5077	0.9344	0.5006	0.5116	0.5049	0.3804	0.4935
	0.0075	0.9961	0.9731	0.9749	0.9824	0.9719	0.9680	0.5003	0.5519	0.5044	0.4974	0.5051	1.0000	0.5000
	0.005	1.0032	1.0023	0.9928	0.9946	0.9699	0.5014	0.5050	0.3417	0.5071	0.9798	0.5002	0.7831	0.5021
	0.0025	0.9999	1.0061	1.0148	1.0467	1.0001	0.8940	0.5010	0.5445	0.5045	0.9164	0.5008	0.0869	0.5128
	0.001	0.9626	0.9712	0.9756	0.9958	1.0020	0.1226	0.4885	0.1162	0.4883	0.5843	0.4959	0.6434	0.4965
	0.00075	0.9927	1.0132	1.0037	1.0103	0.9999	0.6298	0.4964	0.8492	0.5014	0.9541	0.5004	0.7297	0.4974
	0.0005	1.0059	0.9616	1.0007	1.0148	0.9822	0.1040	0.5121	0.9868	0.4999	0.2268	0.5090	0.2904	0.5079
	0.00025	1.0288	1.0155	1.0123	1.0386	0.9874	0.1100	0.5119	0.0811	0.5130	0.5475	0.5045	0.0750	0.5133
	0.0001	0.9785	1.0306	0.9990	1.0171	0.9927	0.5650	0.4957	0.6205	0.5037	0.9074	0.5009	0.4865	0.5052
$d=4$	0.01	0.9828	1.0210	0.9696	0.9957	0.9825	0.9105	0.5008	0.0706	0.5135	0.7666	0.5022	0.7794	0.5021
	0.0075	0.9937	0.9909	0.9948	1.0196	0.9837	0.9355	0.5006	0.6402	0.5035	0.5675	0.5043	0.5133	0.5049
	0.005	0.9970	0.9898	0.9465	1.0015	0.9938	0.9067	0.4991	0.7319	0.5026	0.0621	0.4861	0.8035	0.4981
	0.0025	0.9887	0.9872	1.0226	1.0034	0.9997	0.8361	0.5015	0.8246	0.5017	0.1510	0.5107	0.4005	0.5063
	0.001	1.0171	0.9887	0.9986	1.0243	1.0045	0.2696	0.5082	0.2722	0.4918	0.9347	0.5006	0.7502	0.4976
	0.00075	1.0128	0.9889	0.9711	0.9936	0.9728	0.1720	0.5102	0.8768	0.4988	0.3417	0.5071	0.8209	0.5017
	0.0005	1.0096	0.9630	1.0449	1.0267	0.9811	0.5309	0.5047	0.7116	0.4972	0.1359	0.5111	0.7767	0.5021
	0.00025	0.9809	1.0063	1.0364	1.0243	0.9990	0.8947	0.4990	0.3415	0.5071	0.1735	0.5101	0.2140	0.5093
	0.0001	0.9484	1.0144	1.0115	0.9792	0.9737	0.5817	0.4959	0.3950	0.5063	0.2708	0.5082	0.8562	0.5014
$d=5$	0.01	1.0021	1.0083	0.9985	1.0305	0.9803	0.3913	0.5064	0.0802	0.5130	0.2650	0.5083	0.0184	0.5176
	0.0075	0.9935	0.9912	1.0044	0.9880	0.9831	0.7034	0.5028	0.7561	0.5023	0.2473	0.5086	0.3491	0.4930
	0.005	1.0048	1.0164	1.0047	1.0078	1.0093	0.9020	0.4991	0.2145	0.5093	0.7905	0.5020	0.7843	0.4980
	0.0025	0.9981	1.0176	0.9876	1.0220	1.0015	0.5733	0.4958	0.5393	0.5046	0.3196	0.4926	0.4968	0.5051
	0.001	1.0273	1.0145	0.9767	1.0120	0.9975	0.0960	0.5124	0.3995	0.5063	0.7867	0.4980	0.8510	0.4986
	0.00075	0.9876	1.0085	1.0022	0.9906	1.0023	0.3648	0.4932	0.8394	0.5015	0.9347	0.4994	0.4381	0.4942
	0.0005	1.0105	1.0019	0.9939	1.0071	1.0004	0.3574	0.5069	0.1638	0.5104	0.3812	0.5065	0.5061	0.5050
	0.00025	0.9610	1.0097	1.0192	1.0370	0.9761	0.9183	0.4992	0.1216	0.5115	0.1098	0.5119	0.0135	0.5184
	0.0001	0.9793	1.0492	1.0028	1.0161	0.9857	0.3058	0.4924	0.1979	0.5096	0.7195	0.4973	0.7164	0.5027
$d=8$	0.01	0.9848	1.0163	1.0181	0.9889	0.9957	0.3441	0.4929	0.3646	0.5068	0.4639	0.5055	0.6372	0.4965
	0.0075	0.9996	0.9989	0.9939	0.9712	0.9805	0.3301	0.5073	0.3679	0.5067	0.6421	0.5035	0.4290	0.4941
	0.005	1.0011	0.9986	1.0087	1.0110	1.0097	0.6736	0.4969	0.9909	0.5001	0.6993	0.5029	0.1486	0.5108
	0.0025	0.9959	0.9943	1.0552	1.0282	1.0143	0.4196	0.4940	0.7069	0.4972	0.0856	0.5128	0.9979	0.5000
	0.001	0.9857	0.9872	1.0266	1.0052	0.9978	0.4422	0.4943	0.3432	0.4929	0.5766	0.5042	0.7306	0.4974
	0.00075	1.0087	0.9874	1.0018	1.0043	0.9730	0.4577	0.5055	0.6169	0.4963	0.3710	0.5067	0.6563	0.5033
	0.0005	1.0183	0.9995	0.9717	0.9815	0.9841	0.2968	0.5078	0.1477	0.5108	0.8576	0.4987	0.5847	0.4959
	0.00025	0.9732	0.9693	1.0008	1.0189	0.9999	0.1213	0.4885	0.3881	0.4936	0.6448	0.4966	0.1583	0.5105
	0.0001	1.0012	0.9998	0.9771	1.0006	0.9610	0.0603	0.5140	0.0609	0.5140	0.2538	0.5085	0.0462	0.5149

to 8, all p-values are much less than 0.05, indicating highly significant differences between NNDC-ART and FSCS-ART; while the effect size values range from 0.52 to 0.57, which means that NNDC-ART has the probability of about 52% to 57% that performs better than FSCS-ART. In addition, when comparing NNDC-ART against ART-B-Loc and ART-DC, most of p-values are larger than 0.05 and most of the effect size values are around 0.50 for $1 \leq d \leq 2$, which indicates that there are not significant differences between NNDC-ART and ART-B-Loc and ART-DC. However, when $d \geq 3$, most effect size values are less than 0.50, and most p-values are equal to 0, indicating that NNDC-ART becomes significantly worse than ART-B-Loc and ART-DC.

Overall, compared to FSCS-ART, NNDC-ART has comparable failure-detection effectiveness for the strip patterns. However, NNDC-ART achieves better or similar performances for the block and point patterns, especially when the dimension is high (in this case NNDC-ART outperforms FSCS-ART). Compared with ART-B-Loc and ART-DC, NNDC-ART has similar failure detection effectiveness for strip patterns. However, for the block and point failure region, the differences between NNDC-ART and the compared methods are obvious. In the low dimension of block pattern, NNDC-ART, ART-B-Loc and ART-DC with large λ-value perform similarly, meanwhile NNDC-ART achieves much better F-measure results than ART-DC ($\lambda = 1$). In addition, these algorithms show the slight differences for point failure region in low dimension spaces. However, when the dimension of input domain becomes higher, such as $d = 8$, the failure detection capabilities of ART-DC ($\lambda = 1$) are better than those of other algorithms including the NNDC-ART.

5.2.2. Results of empirical studies

Table 5 presents the F-ratio results for 19 real-life programs, from which we can have the following observations:

Table 4
F-ratio comparisons between NNDC-ART and other ART algorithms for simulations with the *point patterns*.

Dim. (d)	Failure Rate (θ)	F-ratio					Statistical Analysis							
		FSCS-ART	ART-B-Loc	ART-DC ($\lambda=1$)	ART-DC ($\lambda=50$)	NNDC-ART	*vs.* FSCS		*vs.* ART-B-Loc		*vs.* ART-DC ($\lambda=1$)		*vs.* ART-DC ($\lambda=50$)	
							p-value	effect size	p-value	effect size	p-value	effect size	p-value	effect size
$d=1$	0.01	0.9608	0.9544	0.9454	0.9675	0.9735	0.7099	0.5028	0.7291	0.4974	0.3748	0.4934	0.7703	0.5022
	0.0075	0.9436	0.9738	0.9761	0.9428	0.9642	0.2243	0.4909	0.4953	0.4949	0.6547	0.5033	0.0672	0.4864
	0.005	0.9827	0.9572	0.9694	0.9647	0.9621	0.6800	0.5031	0.8332	0.5016	0.5272	0.5047	0.7973	0.5019
	0.0025	0.9559	0.9805	0.9848	0.9531	0.9535	0.4420	0.4943	0.7918	0.5020	0.2405	0.5087	0.5440	0.4955
	0.001	0.9330	0.9463	0.9556	0.9349	0.9696	0.1459	0.4892	0.2552	0.4915	0.3661	0.4933	0.1369	0.4889
	0.00075	0.9720	0.9844	0.9256	0.9913	0.9830	0.6397	0.4965	0.4922	0.4949	0.0022	0.4772	0.2959	0.4922
	0.0005	0.9659	0.9580	0.9601	0.9931	0.9651	0.7683	0.4978	0.6775	0.4969	0.3233	0.4926	0.5152	0.5049
	0.00025	0.9858	0.9786	0.9698	0.9844	0.9643	0.4810	0.5053	0.8418	0.5015	0.4569	0.5055	0.5256	0.5047
	0.0001	0.9671	0.9799	0.9479	0.9312	0.9604	0.2746	0.5081	0.4241	0.5060	0.5942	0.4960	0.1958	0.4904
$d=2$	0.01	0.9904	1.0202	0.9678	0.9902	1.0065	0.3996	0.4937	0.9521	0.5004	0.0067	0.4798	0.1749	0.4899
	0.0075	0.9891	1.0206	0.9427	1.0041	1.0349	0.0828	0.4871	0.3883	0.4936	0.0001	0.4706	0.0356	0.4843
	0.005	1.0078	0.9639	0.9906	0.9958	1.0072	0.6783	0.4969	0.3381	0.4929	0.2635	0.4917	0.7484	0.4976
	0.0025	0.9756	1.0076	0.9716	0.9914	0.9545	0.9688	0.4997	0.1299	0.5113	0.6030	0.5039	0.1719	0.5102
	0.001	0.9886	1.0042	0.9863	0.9954	0.9733	0.2848	0.5080	0.1071	0.5120	0.3783	0.5066	0.8918	0.5010
	0.00075	0.9666	0.9789	0.9690	1.0152	0.9504	0.4524	0.5056	0.4696	0.5054	0.4636	0.5055	0.0632	0.5138
	0.0005	0.9772	0.9811	0.9454	0.9654	0.9991	0.2891	0.4921	0.4093	0.4938	0.0745	0.4867	0.1004	0.4878
	0.00025	0.9924	1.0037	0.9888	0.9980	0.9768	0.1935	0.5097	0.1109	0.5119	0.8119	0.5018	0.3362	0.5072
	0.0001	0.9922	0.9879	1.0006	0.9602	0.9815	0.6939	0.5029	0.4839	0.5052	0.2827	0.5080	0.6684	0.4968
$d=3$	0.01	1.0998	1.1061	0.9605	1.1006	1.1017	0.4325	0.4941	0.8014	0.5019	0.0000	0.4600	0.9760	0.4998
	0.0075	1.0854	1.0920	1.0036	1.0880	1.1054	0.7654	0.4978	0.8680	0.4988	0.0014	0.4763	0.5158	0.4952
	0.005	1.0581	1.0402	0.9878	1.0892	1.1039	0.0444	0.4850	0.0158	0.4820	0.0000	0.4627	0.7194	0.4973
	0.0025	1.0205	1.0185	1.0165	1.0479	1.0398	0.2745	0.4919	0.2498	0.4914	0.0272	0.4835	0.8522	0.4986
	0.001	1.0335	1.0331	0.9675	1.0340	1.0505	0.3540	0.4931	0.6460	0.4966	0.0063	0.4796	0.7150	0.4973
	0.00075	1.0030	1.0297	0.9779	1.0515	1.0166	0.4028	0.4938	0.6183	0.4963	0.1096	0.4881	0.5921	0.5040
	0.0005	1.0139	1.0136	0.9577	1.0242	1.0450	0.1168	0.4883	0.4363	0.4942	0.0003	0.4733	0.1590	0.4895
	0.00025	0.9954	1.0289	0.9800	1.0031	0.9966	0.8494	0.5014	0.4827	0.5052	0.2076	0.4906	0.6624	0.4967
	0.0001	0.9892	0.9724	0.9904	1.0037	0.9982	0.6686	0.4968	0.4791	0.4947	0.4945	0.4949	0.8759	0.4988
$d=4$	0.01	1.2958	1.2235	0.9801	1.2677	1.3212	0.2021	0.4905	0.0053	0.4792	0.0000	0.4078	0.1300	0.4887
	0.0075	1.2855	1.2200	0.9841	1.2297	1.3083	0.1000	0.4877	0.0001	0.4716	0.0000	0.4065	0.0019	0.4769
	0.005	1.2140	1.1779	0.9571	1.2471	1.2231	0.1811	0.4900	0.0415	0.4848	0.0000	0.4218	0.7277	0.4974
	0.0025	1.1834	1.1348	0.9743	1.1661	1.1512	0.4939	0.5051	0.1896	0.4902	0.0000	0.4455	0.3677	0.4933
	0.001	1.1416	1.1076	0.9637	1.0848	1.1185	0.9977	0.5000	0.4109	0.4939	0.0000	0.4512	0.1137	0.4882
	0.00075	1.1246	1.1226	0.9745	1.0950	1.1090	0.6784	0.4969	0.6434	0.4965	0.0000	0.4549	0.5667	0.4957
	0.0005	1.1004	1.0587	0.9842	1.1030	1.0826	0.8230	0.4983	0.1705	0.4898	0.0000	0.4665	0.5527	0.4956
	0.00025	1.1186	1.0323	0.9737	1.0664	1.0331	0.0162	0.5179	0.7989	0.4981	0.0025	0.4780	0.3017	0.5077
	0.0001	1.0513	1.0371	0.9640	1.0549	1.0235	0.5014	0.5050	0.9705	0.4997	0.0058	0.4799	0.6973	0.5029
$d=5$	0.01	1.5332	1.3630	0.9786	1.4307	1.5060	0.6956	0.5029	0.0000	0.4686	0.0000	0.3684	0.0014	0.4762
	0.0075	1.5025	1.2940	0.9683	1.4055	1.4590	0.1580	0.5105	0.0000	0.4628	0.0000	0.3765	0.0083	0.4803
	0.005	1.4694	1.2867	1.0176	1.3907	1.4322	0.4215	0.5060	0.0000	0.4659	0.0000	0.3930	0.0203	0.4827
	0.0025	1.3864	1.2131	0.9794	1.3223	1.3433	0.6632	0.5032	0.0000	0.4667	0.0000	0.4032	0.1280	0.4887
	0.001	1.3020	1.1985	1.0079	1.2307	1.2491	0.7130	0.5027	0.0017	0.4766	0.0000	0.4245	0.1966	0.4904
	0.00075	1.2901	1.1721	0.9726	1.2106	1.2347	0.2677	0.5083	0.0117	0.4812	0.0000	0.4231	0.9424	0.4995
	0.0005	1.2355	1.1360	0.9950	1.1722	1.1755	0.2111	0.5093	0.0266	0.4835	0.0000	0.4458	0.6536	0.4967
	0.00025	1.2221	1.1150	1.0005	1.1504	1.1411	0.0054	0.5207	0.1004	0.4878	0.0000	0.4545	0.6509	0.4966
	0.0001	1.1777	1.0841	0.9616	1.1237	1.1308	0.5319	0.5047	0.0124	0.4814	0.0000	0.4429	0.1069	0.4880
$d=8$	0.01	2.7832	1.9062	1.0009	1.3682	2.3639	0.0000	0.5304	0.0000	0.4156	0.0000	0.2402	0.0000	0.3181
	0.0075	2.5962	1.8309	0.9934	1.2981	2.2499	0.0070	0.5201	0.0000	0.4151	0.0000	0.2451	0.0000	0.3101
	0.005	2.4537	1.7194	1.0251	1.3371	2.0556	0.0024	0.5342	0.0000	0.4266	0.0000	0.2694	0.0000	0.3319
	0.0025	2.3161	1.7203	1.0092	1.3940	1.8642	0.0000	0.5544	0.0000	0.4634	0.0000	0.3043	0.0000	0.3738
	0.001	2.0506	1.5465	1.0120	1.5393	1.7003	0.0000	0.5531	0.0000	0.4686	0.0000	0.3449	0.0000	0.4368
	0.00075	2.0589	1.5353	0.9839	1.4878	1.6620	0.0000	0.5609	0.0002	0.4727	0.0000	0.3517	0.0000	0.4436
	0.0005	1.9730	1.4827	0.9894	1.5471	1.5765	0.0000	0.5605	0.0123	0.4813	0.0000	0.3640	0.0031	0.4779
	0.00025	1.8662	1.4109	1.0030	1.4780	1.5776	0.0000	0.5480	0.0013	0.4760	0.0000	0.3725	0.0395	0.4847
	0.0001	1.7911	1.3227	0.9904	1.3137	1.5828	0.0000	0.5383	0.0000	0.4534	0.0000	0.3741	0.0000	0.4666

Table 5
F-ratio comparisons between NNDC-ART and other ART algorithms for real-life programs.

Program	F-ratio					Statistical Analysis							
	FSCS-ART	ART-B-Loc	ART-DC ($\lambda=1$)	ART-DC ($\lambda=50$)	NNDC-ART	*vs.* FSCS		*vs.* ART-B-Loc		*vs.* ART-DC ($\lambda=1$)		*vs.* ART-DC ($\lambda=50$)	
						p-value	effect size	p-value	effect size	p-value	effect size	p-value	effect size
airy	0.5761	0.5007	0.5742	0.3684	0.5432	0.0008	0.5249	0.0014	0.4762	0.0122	0.5187	0.0000	0.3659
bessj0	0.6087	0.5908	0.7332	0.4449	0.5989	0.0147	0.5182	0.7994	0.5019	0.0000	0.5636	0.0000	0.4021
erfcc	0.5971	0.5336	0.6284	0.3962	0.5630	0.0013	0.5239	0.0768	0.4868	0.0000	0.5373	0.0000	0.3757
probks	0.5528	0.5635	0.6942	0.4090	0.5717	0.6051	0.5039	0.2914	0.5079	0.0000	0.5629	0.0000	0.3912
tanh	0.5767	0.5573	0.6744	0.4174	0.5568	0.0013	0.5240	0.2510	0.5086	0.0000	0.5627	0.0000	0.4018
bessj	0.5735	0.6168	0.9669	0.7354	0.7912	0.0000	0.4316	0.0000	0.4314	0.0000	0.5541	0.3239	0.4926
gammq	0.9115	0.9067	0.9700	0.8571	0.9066	0.4676	0.5054	0.9976	0.5000	0.0193	0.5174	0.2350	0.4911
sncndn	1.0269	1.0473	1.0271	1.0425	1.0195	0.3830	0.5065	0.7560	0.5023	0.7009	0.5029	0.9263	0.5007
golden	0.9985	0.9629	1.0103	0.9430	0.9796	0.3572	0.5069	0.6693	0.5032	0.1166	0.5117	0.8250	0.4984
plgndr	0.5770	0.7138	0.9793	0.5792	0.3650	0.0000	0.6237	0.0000	0.6605	0.0000	0.7365	0.0000	0.6031
cel	0.5254	0.5414	1.0377	0.8979	0.9400	0.0000	0.3619	0.0000	0.3651	0.0007	0.5251	0.1131	0.4882
el2	0.4932	0.5597	0.9730	0.5636	0.4383	0.0000	0.5317	0.0000	0.5548	0.0000	0.6916	0.0000	0.5554
calDay	0.6983	0.5514	0.8369	0.4085	0.6664	0.7059	0.5028	0.0000	0.4376	0.0000	0.5496	0.0000	0.3732
complex	1.0416	1.0129	0.9737	0.9869	1.0198	0.7782	0.5021	0.5008	0.4950	0.0292	0.4837	0.0786	0.4869
pntLinePos	1.0541	1.0221	0.9470	0.9611	1.0233	0.3682	0.5067	0.4672	0.5054	0.0019	0.4769	0.0303	0.4839
triangle	0.9962	1.0153	1.0221	1.0146	0.9621	0.4052	0.5062	0.1527	0.5107	0.0372	0.5155	0.0609	0.5140
line	0.9793	0.9611	0.9708	0.9843	1.0183	0.2559	0.4915	0.0435	0.4849	0.1817	0.4900	0.1861	0.4901
pntTrianglePos	0.6665	0.5228	0.6776	0.4154	0.5982	0.0000	0.5371	0.0001	0.4700	0.0000	0.5385	0.0000	0.4222
twoLinesPos	1.0701	0.8784	0.8033	0.7979	0.9840	0.0101	0.5192	0.0000	0.4637	0.0000	0.4430	0.0000	0.4350

(1) Compared with RT, NNDC-ART achieves comparable performances of fault detection effectiveness for 5 programs `sncndn`, `complex`, `pntLinePos`, `line`, and `twoLinesPos`. However, for other 14 programs, NNDC-ART is much better than RT.

(2) As compared to FSCS-ART, NNDC-ART has worse F-ratio performances for the programs `probks`, `bessj`, `cel`, and `line`. However, the differences between FSCS-ART and NNDC-ART are not significant for the programs `probks` and `line`, because the corresponding p-values are much higher than 0.05. For the other 15 programs, NNDC-ART achieves better or similar F-ratio results to FSCS-ART. Among them, there are 8 real-life programs with the p-values much less than 0.05 (which indicates that NNDC-ART and FSCS-ART have highly significant differences), and the corresponding effect size values range from about 0.52 to 0.63.

(3) Compared with ART-B-Loc, NNDC-ART shows better or similar F-ratio results for most programs. Additionally, it can be seen that the comparisons between NNDC-ART and ART-DC are different for different λ values (used in ART-DC). In detail, when $\lambda = 1$, NNDC-ART is similar or better than ART-DC in most cases; however, when $\lambda = 50$, NNDC-ART is similar or worse than ART-DC. In other words, different λ values may incur different testing results; while NNDC-ART does not suffer from this challenge.

Here, we briefly analyze the reasons to explain why NNDC-ART is worse than FSCS-ART with highly significant differences for the programs `cel` and `bessj`. A closer inspection was conducted on the failure regions of these two programs. We found that the failure region of the program `cel` is a strip along the entire edge of the input domain; while the failure region of the program `bessj` is a shape of two connected triangles with a larger number of failure-causing inputs that are centered around a boundary. As discussed in Section 2.2, FSCS-ART suffers from the boundary effect problem, resulting in more test cases around the boundary of the input domain. For the programs `cel` and `bessj`, therefore, FSCS-ART can quickly identify failure-causing inputs around the boundary. As we know, NNDC-ART aims at evenly distributing test cases over the input domain, to alleviate the boundary effect problem. As illustrated in Section 5.1, NNDC-ART achieves better test case distribution than FSCS-ART, however, it has worse performances to detect the first failure with the failure region around the boundary of the input domain. Nevertheless, it still performs better than RT for the programs `cel` and `bessj`.

Overall, NNDC-ART has similar or better performances than FSCS-ART in most real-life programs, in terms of fault detection effectiveness.

Summary of Answers to RQ2: *Both simulations and empirical studies demonstrate that NNDC-ART achieves comparable or better performances of fault detection than FSCS-ART in most cases. As compared with other ART algorithms including ART-DC and ART-B-Loc, NNDC-ART could obtain similar or even better testing performances in some real-life programs.*

5.3. Answers to RQ3: comparisons of efficiency

5.3.1. Results of simulations

Table 6 provides the detailed comparisons of execution time of NNDC-ART against other ART algorithms for different dimensions, when generating the same number of test cases N.

Based on the above experimental results, we can have the following observations:

(1) It can be seen that NNDC-ART has much lower execution time than FSCS-ART, regardless of the dimension and the number of test cases.

(2) From this table, when the number of generated test cases is small, the execution time improvement of NNDC-ART over FSCS-ART is not large. However, with the increase of N, it is evident that NNDC-ART reduces much more computational time than FSCS-ART. The statistical analysis also validates these observations.

(3) For the 5-dimensional and 8-dimensional input domains, when the number of test cases is equal to 100, the computational time of NNDC-ART is larger than that of FSCS-ART. The main reason for this is that: When d is large, NNDC-ART is very similar to FSCS-ART, according to the distance calculation, because all previously executed test cases may have a high probability to be located in the adjacent subdomains. Compared with FSCS-ART, however, NNDC-ART requires to do the partitioning schema and test case assignment, which also brings additional execution time.

(4) Compared with ART-B-Loc, when $1 \leq d \leq 4$, most of the effect size values are greater than 0.50 and the corresponding p-values are all 0, which implies that NNDC-ART is significantly better than ART-B-Loc. when $d = 5$ and $d = 8$, the time overhead of NNDC-ART is larger than that of ART-B-Loc.

(5) For ART-DC, its time overhead is significantly influenced by the parameter selection of λ. When λ increases the time overhead increases accordingly. When $d = 1$ and $N \geq 5000$, NNDC-ART is significantly better than ART-DC. When $d \geq 2$, the time overhead of NNDC-ART is greater than that of ART-DC for $\lambda = 1$. However, for ART-DC ($\lambda = 50$), when $d = 2$ and $N \geq 1000$, NNDC-ART still has a significant efficiency advantage. In other words, NNDC-ART is more efficient than ART-DC when generating a larger number of test cases in low dimensions.

Overall, it is evident that NNDC-ART requires much less computational time than FSCS-ART to generate the same number of test cases, especially when more test cases are required to be generated. Compared with ART-B-Loc and ART-DC, NNDC-ART still has advantages, especially when the dimension is low and a larger number of test cases is required to be generated.

5.3.2. Empirical studies results

Table 7 further reports the average time to detect the first failure (i.e., F-time) for all object programs. From the results, we can observe that the F-time of NNDC-ART is much less than that of FSCS-ART. From this table, it can be also observed

Table 6
Comparisons of test case execution time between NNDC-ART and other ART algorithms.

Dim. (d)	Number of Test Cases (N)	Execution Time (ms)					Statistical Analysis							
		FSCS-ART	ART-B-Loc	ART-DC ($\lambda = 1$)	ART-DC ($\lambda = 50$)	NNDC-ART	*vs.* FSCS		*vs.* ART-B-Loc		*vs.* ART-DC ($\lambda = 1$)		*vs.* ART-DC ($\lambda = 50$)	
							p-value	effect size	p-value	effect size	p-value	effect size	p-value	effect size
$d = 1$	100	0.64	0.55	0.25	0.50	0.52	0.0000	0.5416	0.4451	0.5050	0.0000	0.3748	0.2015	0.4917
	200	2.08	1.06	0.49	1.04	0.89	0.0000	0.9205	0.0000	0.5717	0.0000	0.3238	0.0000	0.5745
	500	12.34	2.58	1.19	2.63	1.24	0.0000	0.9997	0.0000	0.9408	0.0000	0.4783	0.0000	0.9452
	1000	53.40	5.23	2.43	5.33	2.49	0.0000	1.0000	0.0000	0.9938	0.0041	0.4817	0.0000	0.9936
	2000	233.97	10.93	4.99	10.75	5.08	0.0000	1.0000	0.0000	0.9979	0.1060	0.4906	0.0000	0.9977
	5000	1560.30	34.25	15.98	27.41	14.83	0.0000	1.0000	0.0000	0.9993	0.0000	0.8251	0.0000	0.9983
	10000	7652.72	79.64	35.48	58.07	31.73	0.0000	1.0000	0.0000	0.9993	0.0000	0.9286	0.0000	0.9989
	15000	19010.10	124.21	49.42	93.92	45.04	0.0000	1.0000	0.0000	0.9996	0.0000	0.9219	0.0000	0.9991
	20000	34119.77	187.35	82.02	129.21	68.97	0.0000	1.0000	0.0000	0.9997	0.0000	0.9651	0.0000	0.9987
$d = 2$	100	0.96	0.81	0.40	0.61	0.89	0.0000	0.5541	0.4141	0.5047	0.0000	0.3117	0.0000	0.4025
	200	3.27	1.63	0.73	1.35	1.54	0.0000	0.9220	0.0000	0.6096	0.0000	0.2441	0.0014	0.4801
	500	18.55	4.06	1.93	3.15	3.95	0.0000	0.9679	0.0000	0.7228	0.0000	0.6263	0.0000	0.6830
	1000	80.54	8.35	3.62	6.60	6.42	0.0000	1.0000	0.0000	0.6360	0.8457	0.4986	0.0000	0.5993
	2000	336.51	17.55	8.14	12.78	11.69	0.0000	1.0000	0.0000	0.9328	0.0000	0.2721	0.0000	0.3866
	5000	2321.39	55.74	25.42	33.18	31.90	0.0000	1.0000	0.0000	0.9773	0.0000	0.1881	0.0000	0.7454
	10000	11456.88	132.37	49.21	74.08	62.29	0.0000	1.0000	0.0000	0.9938	0.0000	0.2558	0.0000	0.8609
	15000	26320.95	209.72	69.63	120.50	94.07	0.0000	1.0000	0.0000	0.9944	0.0000	0.0093	0.0000	0.9196
	20000	45533.31	307.87	137.26	157.02	154.68	0.0000	1.0000	0.0000	0.9938	0.0000	0.1851	0.0000	0.7135
$d = 3$	100	1.20	1.07	0.57	0.70	1.45	0.0000	0.5416	0.0000	0.4029	0.0000	0.2191	0.0000	0.2648
	200	4.29	2.15	1.03	1.41	2.52	0.0000	0.9205	0.0185	0.5148	0.0000	0.1263	0.0000	0.2591
	500	25.02	5.51	2.41	3.92	4.26	0.0000	0.9997	0.0000	0.8921	0.0000	0.0226	0.0000	0.4194
	1000	105.45	11.43	5.57	7.11	9.08	0.0000	1.0000	0.0000	0.8866	0.0000	0.0026	0.0000	0.0415
	2000	426.75	24.55	10.58	15.01	17.57	0.0000	1.0000	0.0000	0.9408	0.0000	0.0002	0.0000	0.0823
	5000	3170.80	80.15	37.62	39.32	57.61	0.0000	1.0000	0.0000	0.9404	0.0000	0.0001	0.0000	0.0016
	10000	14833.65	192.82	79.80	79.86	111.96	0.0000	1.0000	0.0000	0.9906	0.0000	0.0004	0.0000	0.0009
	15000	32737.16	306.63	118.42	131.21	169.55	0.0000	1.0000	0.0000	0.9962	0.0000	0.0000	0.0000	0.0019
	20000	56277.13	447.61	154.23	191.34	229.49	0.0000	1.0000	0.0000	0.9982	0.0000	0.0000	0.0000	0.0042
$d = 4$	100	1.29	1.35	0.68	0.83	1.46	0.0000	0.5541	0.0925	0.5104	0.0000	0.2350	0.0000	0.2856
	200	4.38	2.78	1.26	1.58	3.16	0.0000	0.9220	0.0861	0.4886	0.0000	0.0524	0.0000	0.1133
	500	26.59	7.08	3.82	4.32	6.98	0.0000	0.9679	0.0000	0.5925	0.0000	0.0017	0.0000	0.0025
	1000	117.98	14.92	6.96	9.52	13.17	0.0000	1.0000	0.0000	0.9325	0.0000	0.0009	0.0000	0.0025
	2000	480.87	32.50	13.27	16.63	27.90	0.0000	1.0000	0.0000	0.9483	0.0000	0.0006	0.0000	0.0003
	5000	3554.79	105.29	56.97	43.22	106.81	0.0000	1.0000	0.0011	0.4759	0.0000	0.0009	0.0000	0.0000
	10000	15239.32	251.80	136.80	107.38	205.42	0.0000	1.0000	0.0000	0.9717	0.0000	0.0019	0.0000	0.0000
	15000	33191.86	404.37	210.76	178.39	309.65	0.0000	1.0(00	0.0000	0.9891	0.0000	0.0024	0.0000	0.0000
	20000	57404.86	592.20	279.45	216.66	419.51	0.0000	1.0000	0.0000	0.9960	0.0000	0.0020	0.0000	0.0000
$d = 5$	100	1.46	1.59	0.93	0.97	2.23	0.0000	0.4530	0.0000	0.2913	0.0000	0.0553	0.0000	0.0595
	200	4.96	3.26	1.65	1.78	4.46	0.0000	0.8674	0.0000	0.1952	0.0000	0.0016	0.0000	0.0014
	500	30.85	8.54	3.81	4.55	10.59	0.0000	0.9997	0.0000	0.0235	0.0000	0.0003	0.0000	0.0000
	1000	133.73	18.15	7.45	10.17	26.30	0.0000	1.0000	0.0000	0.0012	0.0000	0.0003	0.0000	0.0000
	2000	543.43	39.34	26.33	22.97	64.62	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	5000	4117.31	132.39	60.95	48.34	159.08	0.0000	1.0000	0.0000	0.0057	0.0000	0.0000	0.0000	0.0000
	10000	17135.34	320.38	116.11	97.40	345.67	0.0000	1.0000	0.0000	0.0763	0.0000	0.0000	0.0000	0.0000
	15000	37457.68	522.69	168.22	154.67	574.12	0.0000	1.0000	0.0000	0.0343	0.0000	0.0000	0.0000	0.0000
	20000	65155.10	757.31	216.44	217.03	844.49	0.0000	1.0000	0.0000	0.0130	0.0000	0.0000	0.0000	0.0000
$d = 8$	100	2.20	2.43	1.18	1.59	6.49	0.0000	0.4392	0.0000	0.0008	0.0000	0.0000	0.0000	0.0009
	200	7.65	5.02	2.29	2.71	16.53	0.0000	0.8413	0.0000	0.0005	0.0000	0.0000	0.0000	0.0003
	500	48.08	13.11	21.74	6.14	102.69	0.0000	0.9990	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	1000	200.36	28.13	32.85	12.17	227.60	0.0000	0.9997	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	2000	855.44	63.52	54.98	25.40	496.10	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	5000	6393.71	218.03	120.73	74.31	1388.51	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	10000	24443.66	529.74	226.32	187.72	3080.13	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	15000	54146.05	867.39	326.19	360.55	5046.91	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	20000	95641.89	1254.17	420.82	435.49	7319.16	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 7
F-time comparisons between NNDC-ART and other ART algorithms for real-life programs.

Program	F-time (ms)					Statistical Analysis							
	FSCS-ART	ART-B-Loc	ART-DC ($\lambda = 1$)	ART-DC ($\lambda = 50$)	NNDC-ART	*vs.* FSCS		*vs.* ART-B-Loc		*vs.* ART-DC ($\lambda = 1$)		*vs.* ART-DC ($\lambda = 50$)	
						p-value	effect size	p-value	effect size	p-value	effect size	p-value	effect size
airy	60.84	7.50	3.16	4.48	4.57	0.0000	0.8773	0.0000	0.7165	0.0000	0.6488	0.0000	0.6724
bessj0	16.92	4.51	1.96	2.98	2.73	0.0000	0.8355	0.0000	0.7779	0.0000	0.7138	0.0000	0.7428
erfcc	94.97	9.09	3.44	5.98	5.49	0.0000	0.8842	0.0000	0.6953	0.0000	0.6071	0.0000	0.6477
probks	1428.40	54.98	51.20	37.17	49.33	0.0000	0.7374	0.0000	0.5473	0.0032	0.5219	0.0000	0.4155
tanh	9.24	3.43	1.37	1.92	2.43	0.0000	0.8169	0.0000	0.7848	0.0000	0.6843	0.0000	0.7275
bessj	31.86	8.74	5.80	9.20	7.14	0.0000	0.7378	0.0000	0.6462	0.0000	0.6002	0.0000	0.6543
gammq	230.93	18.08	6.99	14.63	10.31	0.0000	0.8116	0.0000	0.6564	0.1284	0.5112	0.0000	0.6289
sncndn	73.25	10.53	3.50	8.80	5.80	0.0000	0.8028	0.0000	0.7016	0.0000	0.6003	0.0000	0.6818
golden	1055.06	66.43	31.75	58.20	55.00	0.0000	0.7438	0.0000	0.5504	0.0000	0.3903	0.0122	0.5187
plgndr	631.59	51.41	35.28	37.11	20.17	0.0000	0.8045	0.0000	0.7136	0.0000	0.6550	0.0000	0.6252
cel	798.39	34.06	23.80	47.34	64.94	0.0000	0.7160	0.0000	0.3786	0.0000	0.2996	0.0000	0.4405
el2	189.97	19.14	10.04	13.36	12.36	0.0000	0.7570	0.0000	0.6217	0.0344	0.5156	0.0000	0.5760
calDay	379.66	20.58	14.75	11.27	47.75	0.0000	0.6597	0.0000	0.3399	0.0000	0.2886	0.0000	0.2375
complex	464.97	32.53	13.73	22.41	75.82	0.0000	0.6295	0.0000	0.3473	0.0000	0.1639	0.0000	0.2332
pntLinePos	643.15	40.99	18.89	30.09	101.72	0.0000	0.6519	0.0000	0.3552	0.0000	0.1664	0.0000	0.2422
triangle	658.78	41.44	22.52	34.85	94.64	0.0000	0.6474	0.0000	0.3614	0.0000	0.1897	0.0000	0.2646
line	4130.55	135.24	64.04	294.48	1054.46	0.0000	0.5620	0.0000	0.1368	0.0000	0.0848	0.0000	0.0969
pntTrianglePos	9936.54	172.86	64.15	275.39	1392.52	0.0000	0.6467	0.0000	0.1419	0.0000	0.0745	0.0000	0.0778
twoLinesPos	22126.03	370.06	77.81	2322.60	2652.26	0.0000	0.6644	0.0000	0.1340	0.0000	0.0377	0.0000	0.1423

that the F-time improvements of NNDC-ART range from 74% to 97% over FSCS-ART, which means that NNDC-ART highly improves the testing efficiency. In addition, the statistical results totally validate these observations, because all p-values are equal to 0 (indicating that the differences between NNDC-ART and FSCS-ART are highly significant), and the effect size values are much larger than 0.50.

Compared with both ART-B-Loc and ART-DC, NNDC-ART has the lower time overhead in program `plgndr`. Moreover, the statistical analysis shows that for the first ten programs NNDC-ART and other ART algorithms most effect size values are greater than 0.50 and most p-values are less than 0.05, which means that NNDC-ART achieves significantly better performances than ART-B-Loc and ART-DC. For the remaining nine programs (with the dimension $d > 3$), most of the effect size values are less than 0.5 and p-values are less than 0.05, which means that for these programs NNDC-ART performs significantly worse than ART-B-Loc and ART-DC. The main reason for this is that: since the number of adjacent regions to be searched is larger than that of ART-B-Loc (such difference becomes more obvious with the increase of the dimension), NNDC-ART may require more time to find the first failure.

Overall, NNDC-ART requires much less time than FSCS-ART to detect the first failure, irrespective of real-life programs. Compared with ART-B-Loc and ART-DC, NNDC-ART has better efficiency for the programs with low dimensions; but achieving worse performances for the programs with high dimensions.

Summary of Answers to RQ3: *Compared with FSCS-ART, both simulations and empirical studies show that NNDC-ART is much more efficient than FSCS-ART. In addition, compared with ART-B-Loc and ART-DC, NNDC-ART may have better efficiency for low dimensional input domains.*

To Conclude: Our NNDC-ART approach not only achieves better test case distribution than FSCS-ART, but also delivers better or comparable performance of fault detection in most cases. In addition, NNDC-ART not only requires much less computational time to generate the same number of test cases than FSCS-ART, and but also needs much less time to identify the first failure. In other words, NNDC-ART is more cost-effective than FSCS-ART.

Moreover, no ART algorithm always achieves best testing performances among NNDC-ART, ART-B-Loc, and ART-DC. For ART-DC, higher λ value may provide better fault detection effectiveness but may achieve higher computational overhead; otherwise, the smaller value may effectively reduce the time overhead of FSCS-ART but may reduce testing effectiveness. Unfortunately, there is no guideline about how to choose an appropriate parameter value of λ for ART-DC. By contrast, NNDC-ART reduces the time overhead of test case generation while maintaining the effectiveness of FSCS-ART as possible, without the assignment of parameter values. Meanwhile, NNDC-ART delivers similar or better testing performances with ART-B-Loc for the input domains with a low dimension.

5.4. Threat to validity

In this section, we list some main potential threats to the validity of the experiments.

5.4.1. Threats to construct validity

The construct validity refers to whether or not we have fairly conducted the studies. In this paper, we used the F-measure (or F-ratio) [16] as the evaluation metric to measure fault detection effectiveness of NNDC-ART. As we know, there are another two popular evaluation metrics of fault detection: *E-measure* and *P-measure*. The former refers to the expected number of failures detected; and the latter refers to the probability of detecting at least one failure [17]. It should be noted that both E-measure and P-measure assume that a set of test cases with a size is given. As discussed in [22], different sizes of test sets may bring the significant impact on the E-measure and P-measure values, so F-measure is more appropriate than E-measure and P-measure to be applied to the comparisons of adaptive methods such as ART. Besides, previous studies have consistently demonstrated that ART has better performances than RT in terms of P-measure [22]; while E-measure is even less suitable than P-measure, because multiple failures may be associated with single fault, as observed in [39]. As a result, we adopted the F-measure rather than E-measure or P-measure as the evaluation metric in this study.

5.4.2. Threats to external validity

The external validity refers to what extent our experimental results can be generalized. The first threat to external validity is the design of simulations. In this paper, we conducted simulations using three representative failure patterns with the limited number of dimensions and failure rates. As we know, there are different types of failure patterns with different sizes and dimensions. The second threat to external validity is the selection of real-life programs. Although these programs cover a wide range of failure rates, their sizes are relatively small.

To address these potential threats, additional studies will be conducted in the future, including more types of simulations and more object programs.

5.4.3. Threats to internal validity

Internal validity refers to how well the experiments were done. We have manually cross-validated our source code on different examples, and we are confident that the simulation and empirical setups are correct. In addition, each experiment has been repeated at least 3000 times to confirm the observations.

Table 8
Brief descriptions and comparisons among techniques of related work.

	Techniques or Descriptions	Comparisons against NNDC-ART
Reduction of Computational Overhead for ART	*ART through Dynamic Partitioning* (ART-DP) [40] *ART through Iterative Partitioning* (ART-IP) [41] *Quasi-Random Testing* (QRT) [42] *Forgetting* [43]	NNDC-ART is not a standalone ART algorithm but adopted to enhance the effectiveness and efficiency of existing ART algorithms. NNDC-ART does not need to pre-assign a parameter before testing; and discards no information during the process of test case generation.
	Fast Random Border Centroidal Voronoi Tessellations (RBCVTFast) [39]	NNDC-ART is more applicable to generate test cases under adaptive scenarios.
	ART with Divide-and-Conquer (ART-DC) [25]	NNDC-ART may achieve a better test case distribution, and does not require to choose an appropriate value for a parameter. NNDC-ART is more consistent and effective to enhance RT.
	Dynamic Mirror ART (DMART) [30]	Compared with NNDC-ART, DF-FSCS-ART still suffers from the same boundary effect problem as FSCS-ART.
	Distance-aware Forgetting for Fixed-size-Candidate-set ART (DF-FSCS-ART) [44]	NNDC-ART aims at alleviating the boundary effect problem for ART, which is not the focus of KDFC-ART.
	KD-tree-enhanced Fixed-size-Candidate-set ART (KDFC-ART) [34]	
Alleviation of Boundary Effect Problem for ART	*Increase of Test Selection Probability from the Center* [45] [46] [47] [48] [49] [50] *Elimination of the Boundaries* [21] [24] [51] [52]	Compared with NNDC-ART, however, these approaches only focus on the alleviation of the boundary effect the problem rather than the problem of high computational overhead.

6. Related work

As we know, a number of attempts have been made to reduce high computational overhead and to address the boundary effect for ART. In this section, we briefly review these techniques, respectively. Table 8 gives a brief description of the related work, and the brief comparisons between NNDC-ART and other ART techniques.

6.1. Reduction of computational overhead for ART

Many techniques have been proposed to reduce the computational overhead for ART, including *ART through Dynamic Partitioning* (ART-DP) [40], *ART through Iterative Partitioning* (ART-IP) [41], *Quasi-Random Testing* (QRT) [42], *forgetting* [43], *Fast Random Border Centroidal Voronoi Tessellations* (RBCVT-Fast) [39], *ART with Divide-and-Conquer* (ART-DC) [25], *Dynamic Mirror ART* (DMART) [30], *Distance-aware Forgetting for Fixed-size-Candidate-set ART* (DF-FSCS-ART) [44], and *KD-tree-enhanced Fixed-size-Candidate-set ART* (KDFC-ART) [34].

6.1.1. ART-DP

ART-DP [40] has been proposed to deal with the problem of high computational overhead of ART, which contains two versions, namely *ART by Bisection* (ART-B) and *ART by Random Partitioning* (ART-RP). ART-B divides the input domain into equally-sized subdomains by bisecting the longest coordinate. The next test case will be randomly selected from an empty subdomain. If all subdomains have the executed test cases, ART-B will do the next partitioning. Different from ART-B, ART-RP selects the next test case randomly from the largest subdomain, and then uses this test case as the partitioning point to divide the subdomain further. As discussed in [53], the time complexities of ART-B and ART-DP are $O(N)$ and $O(N \log N)$, respectively, when generating N test cases. Although our NNDC-ART approach also makes use of the dynamic partitioning, it is not a standalone ART algorithm but is proposed to enhance the effectiveness and efficiency of existing ART algorithms.

6.1.2. ART-IP

ART-IP [41] first divides the input domain into $p * p$ subdomains, where p is a given constant. To generate the next test case, ART-IP needs to identify a subdomain that satisfies the following two requirements: 1) this subdomain contains no test cases (called *non-empty subdomains*); and 2) it is not adjacent to any non-empty subdomains. Once the partitioning condition is met, the input domain will be divided into $(p+1) * (p+1)$ subdomains, which means that the whole input domain is required to be re-partitioned (this is different against ART-DP). Intuitively, different p values may provide different performances for ART-IP, which also provides a difficulty for testers about the value selection of p. Similar to ART-DP, ART-IP is a standalone ART algorithm, while NNDC-ART can be used to overcome some drawbacks of many existing ART algorithms, such as the high computation overhead and boundary effect of the ART algorithms based on distance (including FSCS-ART and RRT [9]).

6.1.3. QRT

QRT [42] makes use of *quasirandom sequences* [54] to support test case generation for ART. Quasirandom sequences are point sequences with low discrepancy and low dispersion, which generally has a more even distribution over the input domain [19]. As discussed in [42], the time complexity order of QRT is only $O(N)$, which is similar to that of RT, when generating N quasi-random test cases. In other words, QRT requires a low computational cost to achieve a good test case distribution. Different from NNDC-ART, QRT is a standalone ART algorithm.

6.1.4. Forgetting

Intuitively speaking, if the next test case generation is dependent on the previously executed test cases (i.e., $|E|$), its computational overhead could become higher, especially when the size of E becomes larger. Chan et al. [43] proposed

a forgetting strategy, which takes advantage of the constant number M ($M \ll |E|$) of previously executed test cases to generate new test cases. In other words, the forgetting strategy makes the next test case generation independent of $|E|$, resulting in the linear time complexity order. As discussed in [43], however, different M values may provide highly different performances, which provides a challenge about how to assign an "*appropriate*" value to M. Our NNDC-ART approach also uses the concept of forgetting to eliminate distance computations, however, it does not suffer from this challenge. As a consequence, NNDC-ART delivers a more consistent enhancement over RT than the forgetting strategy. In addition, the forgetting strategy discards some previously generated test cases for generating new test cases. However, NNDC-ART fully uses previous test cases while forgetting some distance calculations, which provides no impacts on the selection of new test cases.

6.1.5. RBCVT-fast

Shahbazi et al. [39] have proposed a new testing method by making use of the concept of *Centroidal Voronoi Tessellations*, namely *Random Border Centroidal Voronoi Tessellations* (RBCVT), to achieve random test cases evenly spread over the input domain. Because the order of time complexity of RBCVT is $O(N^2)$ where N is number of test cases, they have developed a fast alternative of RBCVT, i.e., RBCVT-Fast with the time complexity in linear order, by using a novel search algorithm. However, RBCVT-Fast requires a given test set of size N as an input to generate a new test set with the same size, which means that N is a parameter for RBCVT-Fast. In other words, it cannot generate test cases adaptively, which may be not suitable for adaptive testing environments. Compared with RBCVT-Fast, our NNDC-ART is more applicable to generate test cases under adaptive scenarios.

6.1.6. ART-DC

ART-DC is an efficient ART algorithm by applying the concept of divide-and-conquer [25]. It first uses a specific ART algorithm to generate the test cases from the whole input domain. Once the number of test cases reaches a predefined threshold (denoted by λ), the input domain is divided into equal-sized subdomains by bisectionally dividing each dimension, and then test cases are generated from these subdomains independently using the same ART algorithm. The simulation results show that ART-DC can significantly reduce time overhead, while maintaining the comparable failure-detection effectiveness to original ART algorithms.

Intuitively speaking, ART-DC suffers from the following drawbacks: 1) its performances highly depend on the setting of the threshold λ value, i.e., higher λ value may provide better fault detection effectiveness but higher computational overhead (unfortunately no practical guideline is existed to support the selection of λ); and 2) since all subdomains are independent to construct test cases, which may aggravate the boundary effect around the boundaries of subdomains. Although NNDC-ART also makes use of the concept of divide-and-conquer, NNDC-ART may achieve a better test case distribution, because it generates only one test case from the source subdomain by calculating distances against the limited adjacent subdomains. In addition, NNDC-ART does not require to assign an appropriate value for a parameter.

6.1.7. DMART

DMART [30] was proposed to overcome some disadvantages of *Mirror ART* (MART) [55], based on dynamic partitioning. DMART adopts the similar mechanism to ART-DC for dividing the input domain. However, DMART chooses half of subdomains to generate test cases using a specific ART algorithm, and then mirrors these test cases to another half of subdomains for generating the remaining test cases. The order of time complexity of DMART is $O(N)$ to generate N test cases. Similar to ART-DC, DMART also requires a threshold value, and may not alleviate the boundary effect problem. As a result, NNDC-ART may be more consistent and effective to enhance RT.

6.1.8. DF-FSCS-ART

DF-FSCS-ART [44] utilizes the spatial distribution of previously executed test cases to ignore those test cases which are out of "*sight*" of a given candidate. More specifically, the input domain is first divided into $p * p$ subdomains, where p is a predefined parameter. During the generation of test cases, FSCS-ART is applied to the whole input domain, and only previously executed test cases inside the neighboring subdomains of candidates are involved the distance computation process. Once the partitioning conditions are satisfied, a new-round partitioning will be performed. To avoid a large number of test cases in the subdomains, DF-FSCS-ART randomly filters these test cases to keep it within a threshold λ. Intuitively, DF-FSCS-ART faces some challenges: 1) two parameters (p and λ) are required to be set before testing, and different values of these two parameters may lead to highly different performances of DF-FSCS-ART; and 2) with the increase of the number of test cases, more distance calculations are required, especially when λ is large. Moreover, although NNDC-ART and DF-FSCS-ART have the same goal, i.e., the reduction of computational overhead, DF-FSCS-ART still suffers from the same boundary effect problem as FSCS-ART.

6.1.9. KDFC-ART

KDFC-ART [34] was proposed to reduce the high computation of FSCS-ART, based on *K-Dimensional tree* (KD-tree) structure [56]. The first version of KDFC-ART was *Native-KDFC*, which sequentially divides the input space with respect to each dimension to construct a KD-tree. To improve the balance of KD tree, the second version of KDFC-ART was proposed, namely *SemiBal-KDFC*, which prioritizes the splitting based on the spread in each dimension. However, the KD-tree structure suffers

a drawback, i.e., all nodes are required to be traversed in the worst case, especially along with the increase of the dimension (because the number of nodes could significantly increase). As a result, the third version of KDFC-ART was proposed, namely *LimBal-KDFC*, which introduces a upper bound to control the number of traversed nodes in backtracking. Although both NNDC-ART and KDFC-ART focus on the reduction of high computational overhead for ART, the order of time complexity of KDFC-ART is $O(N \log N)$ on the average but $O(N^2)$ in the worst case; while that of NNDC-ART is $O(N)$, where N is the number of test cases to be generated. In addition, NNDC-ART also aims at alleviating the boundary effect problem for ART, which is not the focus of KDFC-ART.

6.2. Alleviation of boundary effect problem for ART

Previous studies have proposed a number of approaches to address the boundary effect problem for ART, which can be divided into the following categories: *Increase of Test Selection Probability from the Center* and *Elimination of the Boundaries*. The former is to increase the probability of test inputs from the center of the input domain; while the latter is to eliminate the concept of boundaries of the input domain. Compared with NNDC-ART, however, these approaches only focus on the alleviation of boundary effect problem rather than the problem of high computational overhead.

6.2.1. Increase of test selection probability from the center

Chen et al. [45] designed a new test profile for the candidate generation process, which guarantees that test inputs in the central part of the input domain have a higher probability to be selected as candidates than those in the boundary part. Kuo et al. [46] adopted a filtering process to enhance original ART algorithms, which enforces test cases with a lower boundary preference. As discussed in [47], a new ART method called *Inverted FSCS-ART* was proposed to invert the boundary/center distribution of FSCS-ART's test cases by mapping some test cases from the boundary to the center of the input domain and vice versa.

Chen et al. [48] first divided the input domain into the same-sized partitions from the boundary to the center of the input domain, and then generated k candidates from the partitions without any test cases, which could evenly spread test cases in the boundary and center of the input domain. Besides, Chen et al. [49] proposed a new approach named *ART by balancing*, which prefers to choose test cases close to the input domain centroid. Mayer [50] first generated test cases from a small region around the center of the input domain, and then extended the region to select the following test cases if no failures were detected.

6.2.2. Elimination of the boundaries

Mayer and Schneckenburger [51] proposed a new distance measure by taking account of same-sized adjacent regions of the input domain to remove its boundaries, where the input domain is regarded as virtually continuous. Similarly, Chen et al. [21] considered the virtually adjacent subdomains for removing the boundaries of the input domain. In addition, Geng and Zhang [24] jointed the boundaries of the input domain for calculating the distance between test cases.

Mayer and Schneckenburger [52] enlarged the original input domain by an enlargement factor on each boundary for ART, which could generate some special test inputs outside the input domain. These inputs are not considered as test cases for executing the programs, because they are only used for the distance calculation. Mayer [57] proposed a technique to randomly translate the input domain, which could be adopted for different ART techniques in order to avoid test cases being selected more frequently close to the boundary of the input domain.

7. Conclusions and future work

Adaptive Random Testing (ART) [9] has been proposed to improve failure-detection capability of *Random Testing* (RT). The *Fixed-Size-Candidate-Set ART* (FSCS-ART) [15] is the most well-know implementation of ART, and has been widely used in practice. Previous studies have demonstrated that FSCS-ART is more effective than RT according to different evaluation criteria such as fault detection capability [15–18], test case distribution [19], and code coverage [20]. However, FSCS-ART suffers from the following two drawbacks: *high computational overhead* and *boundary effect problem* [9].

In this paper, we proposed an alternative ART approach to enhance FSCS-ART based on the concepts of *nearest-neighbor* and *divide-and-conquer*, namely *Nearest-Neighbor Divide-and-Conquer based ART* (NNDC-ART), which aims at reducing the high computational overhead of test case generation and alleviating the boundary effect problem. Our simulations show that NNDC-ART not only requires much less computational overhead than FSCS-ART, but also achieves better test case distribution; while delivering better or comparable fault detection capability. In addition, the results of empirical studies show that NNDC-ART is much more cost-effective than FSCS-ART. Furthermore, we compared NNDC-ART with other ART algorithms including ART-DC and ART-B-Loc. The experimental results show that NNDC-ART provides the competitive failure detection ability to ART-DC and ART-B-Loc in some scenarios.

As we know, there are other ART algorithms that also suffer from the similar weaknesses to FSCS-ART. For example, *Restricted Random Testing* (RRT) [29] is another implementation of ART based on the concept of exclusion, which also requires high computational overhead, and suffers from the boundary effect problem. Therefore, one important direction for future work is on the extension of NNDC-ART to other ART algorithms that have the similar drawbacks.

According to the simulation results, our NNDC-ART approach alleviates the boundary effect problem to some extent as compared to FSCS-ART, especially when the dimension is high. However, NNDC-ART still achieves worse test case distribution and fault detection effectiveness than RT for the high-dimensional input domains. The main reason for this is that it has been suggested that the *high dimension problem*[3] may be influenced (or even caused) by the boundary effect problem, because the center of a high-dimensional input domain has a higher chance of being a failure region than the boundary [49,50]. In other words, compared with RT, NNDC-ART still suffers from the boundary effect problem and high dimension problem. As a result, it would be very promising to develop other new ART approaches for overcoming main challenges of ART.

CRediT authorship contribution statement

Rubing Huang: Conceptualization, Methodology, Writing- Reviewing and Editing, Investigation.
Weifeng Sun: Software, Data curation, Writing - Original draft reparation.
Haibo Chen: Visualization, Formal analysis, Methodology verification.
Chenhui Cui: Proof-reading, Validation.
Ning Yang: 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

The authors would like to thank Junlong Lian for his valuable discussion and comments on this paper. This work is in part supported by the National Natural Science Foundation of China, under grant nos. 61872167 and 61502205. This work is also partly supported by the Science and Technology Program of the Ministry of Housing and Urban-Rural Development of China, under grant no. 2020-S-001, and the Postgraduate Research & Practice Innovation Program of Jiangsu Province, under grant no. KYCX19_1616.

References

[1] A. Orso, G. Rothermel, Software testing: a research travelogue (2000-2014), in: Proceedings of Future of Software Engineering (FOSE'14), 2014, pp. 117–132.
[2] A. Arcuri, M. Iqbal, L. Briand, Random testing: theoretical results and practical implications, IEEE Trans. Softw. Eng. 38 (2) (2012) 258–277.
[3] B.P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of Unix utilities, Commun. ACM 33 (12) (1990) 32–44.
[4] J. Forrester, B. Miller, An empirical study of the robustness of windows NT applications using random testing, in: Proceedings of the 4th USENIX Windows Systems Symposium (WSS'00), 2000, pp. 59–68.
[5] T. Yoshikawa, K. Shimura, T. Ozawa, Random program generator for Java JIT compiler test system, in: Proceedings of the 3rd International Conference on Quality Software (QSIC'03), 2003, pp. 20–24.
[6] J. Regehr, Random testing of interrupt-driven software, in: Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05), 2005, pp. 290–298.
[7] H. Bati, L. Giakoumakis, S. Herbert, A. Surna, A genetic approach for random testing of database systems, in: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07), 2007, pp. 1243–1251.
[8] G.J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, John Wiley & Sons, 2011.
[9] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, X. Xia, A survey on adaptive random testing, IEEE Trans. Softw. Eng. 47 (10) (2021) 2052–2083.
[10] L. White, E. Cohen, A domain strategy for computer program testing, IEEE Trans. Softw. Eng. 6 (3) (1980) 247–257.
[11] P.E. Ammann, J.C. Knight, Data diversity: an approach to software fault tolerance, IEEE Trans. Comput. 37 (4) (1988) 418–425.
[12] G.B. Finelli, NASA software failure characterization experiments, Reliab. Eng. Syst. Saf. 32 (1–2) (1991) 155–169.
[13] P.G. Bishop, The variation of software survival times for different operational input profiles, in: Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS'93), 1993, pp. 98–107.
[14] 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.
[15] 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.
[16] 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.
[17] T.Y. Chen, F.-C. Kuo, R.G. Merkel, On the statistical properties of testing effectiveness measures, J. Syst. Softw. 79 (5) (2006) 591–601.
[18] T.Y. Chen, F.-C. Kuo, Z.Q. Zhou, On favourable conditions for adaptive random testing, Int. J. Softw. Eng. Knowl. Eng. 17 (06) (2007) 805–825.
[19] 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.
[20] T.Y. Chen, F.-C. Kuo, H. Liu, W.E. Wong, Code coverage of adaptive random testing, IEEE Trans. Reliab. 62 (1) (2013) 226–237.
[21] T.Y. Chen, D.H. Huang, T.H. Tse, Z. Yang, An innovative approach to tackling the boundary effect in adaptive random testing, in: Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07), 2007, p. 262a.

[3] The *high dimension problem* is another drawback of ART [9], which indicates that the performance of ART could gradually deteriorate along with the increase of the dimension, due to the *curse of dimensionality* [38].

[22] S. Anand, E.K. Burke, T.Y. Chen, J. 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.
[23] J. Mayer, Adaptive random testing by bisection and localization, in: Proceedings of the 5th International Workshop on Formal Approaches to Software Testing (FATES'05), 2005, pp. 72–86.
[24] J. Geng, J. Zhang, A new method to solve the "boundary effect" of adaptive random testing, in: Proceedings of the 7th International Conference on Educational and Information Technology (ICEIT'10), 2010, pp. V1-298–V1-302.
[25] C. Chow, T.Y. Chen, T. Tse, 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), 2013, pp. 268–275.
[26] F. Chan, T.Y. Chen, I. Mak, Y.-T. Yu, Proportional sampling strategy: guidelines for software testing practitioners, Inf. Softw. Technol. 38 (12) (1996) 775–782.
[27] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Trans. Inf. Theory 13 (1) (1967) 21–27.
[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] K.P. Chan, T.Y. Chen, D. Towey, Restricted random testing: adaptive random testing by exclusion, Int. J. Softw. Eng. Knowl. Eng. 16 (4) (2006) 553–584.
[30] R. Huang, H. Liu, X. Xie, J. Chen, Enhancing mirror adaptive random testing through dynamic partitioning, Inf. Softw. Technol. 67 (2015) 13–29.
[31] ACM, Collected Algorithms from ACM, Association for Computer Machinery, New York, 1980.
[32] W. Press, B.P. Flannery, S.A. Teulolsky, W.T. Vetterling, Numerical Recipes, Cambridge University Press, Cambridge, 1986.
[33] J. Ferrer, F. Chicano, E. Alba, Evolutionary algorithms for the multi-objective test data generation problem, Softw. Pract. Exp. 42 (11) (2012) 1331–1362.
[34] 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.
[35] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2010) 649–678.
[36] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Softw. Test. Verif. Reliab. 24 (3) (2014) 219–250.
[37] A. Vargha, H.D. Delaney, A critique and improvement of the CL common language effect size statistics of Mcgraw and Wong, J. Educ. Behav. Stat. 25 (2) (2000) 101–132.
[38] R. Bellman, Dynamic Programming, Princeton University Press, 1957.
[39] 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.
[40] T.Y. Chen, R. Merkel, P.K. Wong, G. Eddy, Adaptive random testing through dynamic partitioning, in: Proceedings of the 4th International Conference on Quality Software (QSIC'04), 2004, pp. 79–86.
[41] T.Y. Chen, D.H. Huang, Z.Q. Zhou, On adaptive random testing through iterative partitioning, J. Inf. Sci. Eng. 27 (4) (2011) 1449–1472.
[42] T.Y. Chen, R.G. Merkel, Quasi-random testing, IEEE Trans. Reliab. 56 (3) (2007) 562–568.
[43] K.-P. Chan, T.Y. Chen, D. Towey, Forgetting test cases, in: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06), 2006, pp. 485–494.
[44] C. Mao, T.Y. Chen, F.-C. Kuo, Out of sight, out of mind: a distance-aware forgetting strategy for adaptive random testing, Sci. China Inf. Sci. 60 (9) (2017) 092106.
[45] T.Y. Chen, F.-C. Kuo, H. Liu, Application of a failure driven test profile in random testing, IEEE Trans. Reliab. 58 (1) (2009) 179–192.
[46] F.C. Kuo, T.Y. Chen, H. Liu, W.K. Chan, Enhancing adaptive random testing for programs with high dimensional input domains or failure-unrelated parameters, Softw. Qual. J. 16 (3) (2008) 303–327.
[47] F.-C. Kuo, K.Y. Sim, C.-A. Sun, S. Tang, Z.Q. Zhou, Enhanced random testing for programs with high dimensional input domains, in: Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07), 2007, pp. 135–140.
[48] T.Y. Chen, F.-C. Kuo, H. Liu, Distributing test cases more evenly in adaptive random testing, J. Syst. Softw. 81 (12) (2008) 2146–2162.
[49] T.Y. Chen, D.H. Huang, F.-C. Kuo, Adaptive random testing by balancing, in: Proceedings of the 2nd International Workshop on Random Testing (RT'07), 2007, pp. 2–9.
[50] J. Mayer, Towards effective adaptive random testing for higher-dimensional input domains, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO'06), 2006, pp. 1955–1956.
[51] J. Mayer, C. Schneckenburger, Statistical analysis and enhancement of random testing methods also under constrained resources, in: Proceedings of the 4th International Conference on Software Engineering Research and Practice (SERP'06), 2006, pp. 16–23.
[52] J. Mayer, C. Schneckenburger, Adaptive random testing with enlarged input domain, in: Proceedings of the 6th International Conference on Quality Software (QSIC'06), 2006, pp. 251–258.
[53] J. Mayer, C. Schneckenburger, An empirical analysis and comparison of random testing techniques, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06), 2006, pp. 105–114.
[54] J.H. Halton, Algorithm 247: radical-inverse quasi-random point sequence, Commun. ACM 7 (12) (1964) 701–702.
[55] T.Y. Chen, F.-C. Kuo, R. Merkel, S.P. Ng, Mirror adaptive random testing, Inf. Softw. Technol. 46 (15) (2004) 1001–1010.
[56] Q. Du, V. Faber, M. Gunzburger, Centroidal Voronoi tessellations: applications and algorithms, SIAM Rev. 41 (4) (1999) 637–676.
[57] J. Mayer, Adaptive random testing with randomly translated failure region, in: Proceedings of the 1st International Workshop on Random Testing (RT'06), 2006, pp. 70–77.