KDFC-ART: a KD-tree approach to enhancing Fixed-size-Candidate-set Adaptive Random Testing

Chengying Mao, *Member, IEEE*, Xuzheng Zhan, T.H. Tse, *Senior Member, IEEE*, and Tsong Yueh Chen, *Senior Member, IEEE*

Abstract—**Adaptive random testing (ART) was developed as an enhanced version of random testing to increase the effectiveness of detecting failures in programs by spreading the test cases evenly over the input space. However, heavy computation may be incurred. In this paper, three enhanced algorithms for fixed-size-candidate-set ART (FSCS-ART) are proposed based on the *k*-dimensional tree (KD-tree) structure. The first algorithm Naive-KDFC constructs a KD-tree by splitting the input space with respect to every dimension successively in a round-robin fashion. The second algorithm SemiBal-KDFC improves the balance of the KD-tree by prioritizing the splitting according to the spread in each dimension. In order to control the number of traversed nodes in backtracking, the third algorithm LimBal-KDFC introduces an upper bound for the nodes involved. Simulation and empirical studies have been conducted to investigate the efficiency and effectiveness of the three algorithms. The experimental results show that these algorithms significantly reduce the computation time of the original FSCS-ART for low dimensions and for the case of high dimensions with low failure rates. The efficiency of SemiBal-KDFC is better than that of Naive-KDFC when the dimension is no more than 8, but LimBal-KDFC is the most efficient of all three. Although limited backtracking leads only to an approximate nearest neighbor in LimBal-KDFC, its failure-detection effectiveness is, in fact, better than FSCS-ART in high-dimensional input spaces and has no significant deterioration in low-dimensional spaces.**

Index Terms—**Adaptive random testing, KD-tree, test case generation, software testing, computation time, efficiency, effectiveness.**

I. Introduction

SOFTWARE testing has shown great potential in detecting program failures, thus assuring software quality. It is regarded as an essential activity in the software development life-cycle [1], [2]. In particular, random testing (RT) [3] is a basic black-box testing method that can easily be implemented in practice and has a wide variety of applications [4]. It is simple and cost-effective because it does not require any information about the software specification, program code, or likely failure regions. Echoing classical findings [5], [6], Ciupa et al. [7] have pointed out that "Counter to what intuition suggests, random strategies for selecting test inputs have proved remarkably effective". Based on theoretical and simulation studies, Arcuri et al. [8] reported that there are practical circumstances where RT should be recommended as the first option to apply. Currently, RT is commonly used as a baseline technique for empirical studies for new testing approaches.

On the other hand, RT has been criticized for not making use of the clustering of failure regions [9]. Chen et al. have proposed adaptive random testing (ART) to address the issue [10]. The underlying principle is to spread the test cases as evenly as possible over the input space of the program under test. The diversity of test cases can obviously be enhanced [11]. As a result, it can achieve a better failure-detection capability than RT [12].

Quite a number of ART methods have been proposed. Among them, the fixed-size-candidate-set algorithm of ART (FSCS-ART) [13] exhibits better failure-detection effectiveness for programs with numeric inputs and has been widely applied. However, using the direct implementation, this method has to calculate the distances from every candidate to all the executed test cases, thus leading to a quadratic computational overhead. Hence, some doubts about its efficiency have been raised [14]. It is therefore necessary to reduce the computational overhead of FSCS-ART without compromising the effectiveness of failure detection.

As mentioned above, the overhead of FSCS-ART is mainly caused by the distance computation for each candidate to find its nearest neighbor from all the executed test cases. To solve this problem, it is necessary to find a smarter search process for the nearest neighbor of each candidate. Intuitively, a more efficient implementation of FSCS-ART should ignore some of the distance computation when determining the nearest neighbor of each candidate. In essence, the executed test cases that are far away from the candidates can be excluded from the distance computation.

In this paper, we propose a KD-tree approach to improve the efficiency of the FSCS-ART algorithm for programs with numeric inputs. We will call it the KD-tree-enhanced Fixed-size-Candidate-set ART (KDFC-ART).[1] A KD-tree [15], which stands for *k*-dimensional tree, is a space-partitioning structure commonly used in computational geometry and spatial databases [16]. It can eliminate quite a number of points in the space when querying the nearest neighbor for a given point.

C. Mao and X. Zhan are with the School of Software and Communication Engineering, Jiangxi University of Finance and Economics, Nanchang 330013, China (Emails: maochy@jxufe.edu.cn; maochy@yeah.net; zhanxuz@yeah.net).

Corresponding author. T.H. Tse is with the Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong (Email: thtse@cs.hku.hk).

T.Y. Chen is with the Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia (Email: tychen@swin.edu.au).

[1]KDFC is the radio home of *San Francisco Symphony* and *San Francisco Opera*, supporting diversities in performing art. We apologize for any unintended pun.

Thus, the data structure is very helpful in speeding up the nearest neighbor queries in FSCS-ART.

A standard KD-tree is normally used to encode an existing *static* data set as a tree structure. In other words, the data set is defined before the construction of the KD-tree. However, in adaptive random testing, test cases are incrementally generated as the testing progresses. Thus, we need to reformulate the construction method of KD-trees so as to incorporate incrementally added test cases into a dynamically growing tree structure.

In order to realize the incremental construction of KD-trees, we need to develop a strategy for inserting nodes, which determines whether they belong to the left or right branch of a subtree. Essentially, we need a good procedure to split each level of input subspaces into two parts.

a) First, we use a naive strategy to split the input space with respect to every dimension successively in a round-robin manner. However, the resulting tree may induce high overheads in subsequent queries because the arbitrary splitting process may generate some search paths that are excessively long.

b) In order to address the above deficiency, a semi-balanced splitting strategy is proposed. When selecting the next dimension to split, we compute the possible spread of points for every dimension and pick the one with the maximum spread. On the other hand, for efficient incremental selection of test cases, we do not follow the normal practice of using the median point as the splitting location. The balance of the tree is guaranteed as much as possible by an adaptive selection strategy of splitting dimension, in which the spread in each dimension is taken into consideration. The resulting semi-balanced tree can achieve much better balancing effect than the tree produced by the naive strategy.

c) The worst case scenario for finding the nearest neighbor in a semi-balanced tree may involve traversing all the nodes. In particular, for high-dimensional input spaces, only a few nodes are pruned during the search [16]. Thus, an approximate nearest neighbor query may be more efficient. Restricted backtracking is further proposed to control the distance computation cost of nearest neighbor queries. Empirical results show that the effectiveness is not compromised in most cases, but is improved in some cases.

The main contribution of this paper is threefold:

1) To the best of our knowledge, this is the first paper that proposes a KD-tree approach to enhance the performance of adaptive random testing.

2) The proposed algorithms significantly improve the efficiency of the original FSCS-ART for low-dimensional input spaces and for high-dimensional spaces with low failure rates.

3) The proposed Naive-KDFC and SemiBal-KDFC algorithms have the same effectiveness as the original FSCS-ART. The LimBal-KDFC algorithm, in fact, improves the effectiveness of FSCS-ART for high-dimensional input spaces and has no significant deterioration in effectiveness for low-dimensional spaces.

The rest of the paper is organized as follows: Section II summarizes the existing FSCS-ART algorithm and introduces the basic concepts of KD-trees. Section III presents a KD-tree-based framework to enhance FSCS-ART, and then proposes three algorithms for this purpose. A simulation analysis of the algorithms is presented in Section IV, followed by an empirical investigation in Section V. Section VI discusses the threats to validity in our study. Related work is discussed in Section VII. Finally, we conclude the paper and give future research directions in Section VIII.

II. Background

A. The Original FSCS-ART Method

In software testing, information about failure patterns can be used to guide test cases selection. Generally speaking, relatively regular failure-causing input regions and programs with relatively higher failure rates require relatively fewer test cases to expose failures. As usual, the *failure rate* refers to the ratio between the number of failure-causing inputs and the number of all possible inputs of a program [13], [17].[2] Without loss of generality, failure patterns can be classified into three categories according to the failure clustering effects: *block patterns*, *strip patterns*, and *point patterns* [18]. For the block pattern, the failure-causing inputs amalgamate into one or a few contiguous regions in the input space. It describes the failures that are often triggered by computational errors, such as when the computational expression in an assignment statement is incorrect. For the strip pattern, the failure-causing inputs are clustered in the shape of a narrow strip. Its failures are attributed to domain errors [19] and are usually caused by predicate faults in a branch. For the point pattern, the failure-causing inputs are standalone points or small groups of points in the input space.

Previous empirical studies have pointed out that failure-causing inputs are often clustered in contiguous regions [20], [21], [22]. In other words, blocks and strips are the predominant failure patterns. In order to achieve a higher fault-detection probability, test cases need to be far apart, and thus diversified over the input space [10], [11]. The adaptive random testing (ART) approach was developed by Chen et al. [10], [11], [13] based on this concept. It aims to achieve a better even spreading of test cases over the input space than random testing. Various ART algorithms have been developed according to different even-spreading principles. Among them, the fixed-size-candidate-set algorithm of ART (FSCS-ART) is the most popular [13] partly because it was the first one proposed and partly because of its high effectiveness.

[2]The failure rate of a program affects the effectiveness and efficiency of test case selection much more than the program size. It is, therefore, recognized as an important decision factor in the choice of testing techniques. Unfortunately, it is virtually impossible to exhaustively execute all the possible inputs for a nontrivial program to find the failure rate. Furthermore, the exhaustive execution of all inputs defeats the purpose of test case selection. Thus, the exact failure rate as per the formal definition above is only used in research studies. In practice, testers have to rely on an estimated failure rate. Still, the failure rate is useful in practice to answer "what if" questions during the decision-making process for an appropriate testing technique.

FSCS-ART is a selection-based ART method [10] using the *max-min distance criterion*, which is the most common selection criterion. The test case generation process by FSCS-ART can be described as follows: Let E be the set of test cases already generated. Initially, E is empty. The first test case is randomly selected from the input space and added to E. Given a non-empty E, a fixed number of inputs $Cand = \{cand_1, cand_2, \ldots, cand_s\}$ are randomly generated as candidates for the next test case. The candidate $cand_{\text{best}}$ with the *maximum distance from its nearest element* in E is then selected as the next test case, that is:

$$\min_{j=1}^{|E|} dist(cand_{\text{best}}, tc_j) \geq \min_{j=1}^{|E|} dist(cand_k, tc_j) \qquad (1)$$

where tc_j ($j \in \{1, 2, \ldots, |E|\}$) is a test case in E, $cand_{\text{best}}$ and $cand_k$ ($k \in \{1, 2, \ldots, s\}$) are elements of the candidate set $Cand$, and $dist$ refers to the Euclidean distance for numeric inputs. The above process of test case selection is repeated until the termination condition is satisfied. In the rest of this paper, the original FSCS-ART algorithm will simply be referred to as FSCS-ART when there is no ambiguity.

For each candidate, the FSCS-ART algorithm has to compute the distances from all previously executed test cases. Hence, its complexity is $O(\text{FSCS-ART}) = \sum_{j=1}^{n-1} js = O(sn^2)$, where s is the size of the candidate set and n is the number of test cases. In other words, FSCS-ART takes quadratic computation time to generate test cases. Theoretically, when the total size of all failure regions in a program is very small, the number of test cases needed to detect a failure will be huge. In such a case, the FSCS-ART algorithm requires a substantial amount of time to generate test cases [14]. Especially for programs consuming little execution time, this disadvantage will become more prominent and will make FSCS-ART less cost-effective than RT. Therefore, the computation time for test case generation has become an important factor that may restrict the practical application of FSCS-ART.

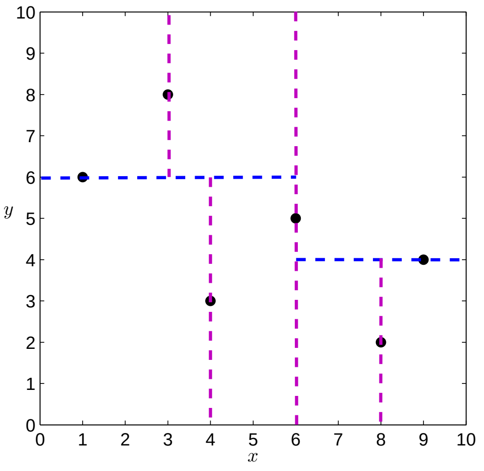
B. Preliminaries of KD-Trees

Given a set of points in a d-dimensional space, a KD-tree (also known as a k-dimensional tree) [15] is a useful data structure for organizing the points through space partitioning.[3] A KD-tree is a special case of binary space partitioning trees such that every node is a d-dimensional point. This data structure helps reduce unnecessary distance computations during the search for the nearest neighbor. It has extensive applications in computational geometry [16], spatial information processing [23], and computer graphics [24], [25].

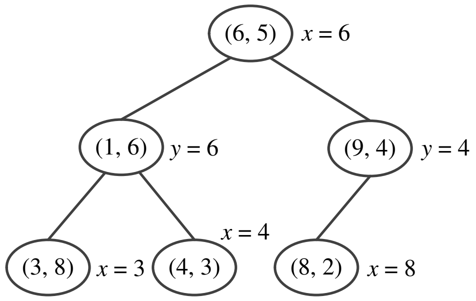
In general, a KD-tree is constructed by recursively splitting a subset at each level into two smaller subsets at the next level, thus progressing from the root to the leaves in a level-by-level manner. Various authors have proposed effective and efficient strategies to select the next dimension to split and the location of the splitting hyperplane for each dimension. Round robin was the original strategy proposed by Bentley [15]. It simply selects the split dimensions one by one sequentially. The maximum range (or maximum spread) is another typical strategy, which was originally proposed by Friedman et al. [26]. Minor variations have also been proposed by others [27] but they are similar in idea to the original.

[3]Following standard practice in KD-tree literature, we use d (rather than k) to denote the number of dimensions in the given space and the KD-tree. In other words, k is never used except in the formal name of KD-trees.

Consider a set of points $\{(1,6), (3,8), (4,3), (6,5), (8,2), (9,4)\}$ in a 2-dimensional space as an example. The KD-tree-based subspace partitioning process is illustrated by Figure 1(a) and the corresponding data structure is illustrated by Figure 1(b). Since the X-coordinates of the set of points have the largest variance, the X-dimension is chosen for the first splitting. The splitting hyperplane is $x = 6$ because 6 is the median of all the X-coordinates. Point $(6,5)$ is the resulting root node. The process is repeated until all the points are represented as nodes, as shown in Figure 1(b).



(a) KD-tree-based partitioning



(b) KD-tree structure

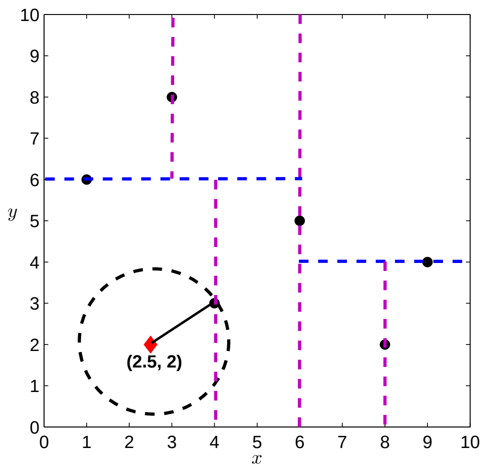
Fig. 1. Example of KD-tree-based partitioning and its storage structure

The main advantage of KD-trees is that it can efficiently perform nearest neighbor queries. Given a tree and a query point, the query process can be described as follows: First, locate the smallest subspace that contains the query point. It is represented by a node in the tree. The localization starts with the root node and moves down the tree recursively. The localization direction (namely, going to the left or right branch) depends on whether the query point's coordinate is not greater than or greater than the value of the current tree node. Once the localization process reaches the node standing for the smallest subspace, the latter is regarded as the temporary nearest neighbor of the query node. The reason why the KD-tree implements an efficient nearest neighbor query is its effective pruning ability. It only needs the distance calculations from the query point to some of the nodes in the tree. During the backtracking process, if the current nearest distance is less than the distance between the query point and the splitting hyperplane associated with an ancestor or sibling node, the other branch can be pruned from backtracking. Owing to effective pruning, we can avoid a great deal of distance computations when locating the nearest neighbor in KD-trees. On the other hand, for some extreme cases in high-dimensional spaces, we still need to traverse nearly all the nodes in the tree.

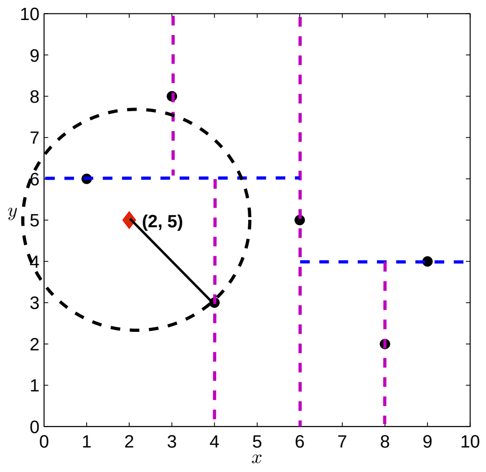
We give two examples in Figure 2 to explain the backtracking process for finding the nearest neighbor. The point set and its KD-tree are the same as those in Figure 1. Suppose the given query point is $Q_1 = (2.5, 2)$, as shown in Figure 2(a). It is easy to locate the corresponding potential parent node in the KD-tree for the query point, namely, the point Q_1 lying in the subregion containing the leaf node $(4, 3)$. Consequently, node $(4, 3)$ is taken to be the temporary nearest neighbor of Q_1. The distance between Q_1 and note $(4, 3)$, which is $\sqrt{3.25}$, is taken to be the current nearest distance. As the next step, consider the parent of node $(4, 3)$. We first compute the distance from the query point Q_1 to the splitting line $y = 6$. Because this distance (which is 4) is larger than the current nearest distance, the associated parent node $(1, 6)$ and all the nodes above the line $y = 6$ can be ignored in the distance computation. Thus, the search backtracks to the parent of node $(1, 6)$, which is the root node $(6, 5)$. Similarly, because the distance from Q_1 to the splitting line $x = 6$ (which is 3.5) is also larger than the current nearest distance, all the nodes on the right of $x = 6$ are pruned from the query. As a result, backtracking will be terminated.

The query point $Q_2 = (2, 5)$ in Figure 2(b) illustrates a more complex situation. First, Q_2 is identified to be within the subregion containing the leaf node $(4, 3)$. Consequently, node $(4, 3)$ is taken to be the temporary nearest neighbor with a current nearest distance of $2\sqrt{2}$. Next, consider the splitting line $y = 6$ associated with the parent of node $(4, 3)$. Because the distance from Q_2 to the splitting line $y = 6$ (which is 1) is smaller than the current nearest distance, the parent and sibling nodes of $(4, 3)$ should also be considered during backtracking. First, the distance between Q_2 and node $(1, 6)$ is further calculated to be $\sqrt{2}$. This distance is less than that between Q_2 and node $(4, 3)$, and hence $(1, 6)$ is updated as the temporary nearest neighbor. Subsequently, the distance from Q_2 to the splitting line $x = 3$ is calculated. Since the result is 1 and less than the current nearest distance, the distance between Q_2 and node $(3, 8)$ will then be computed. The latter is $\sqrt{10}$ and larger than the distance between Q_2 and node $(1, 6)$. Hence, $(1, 6)$ is still the temporary nearest neighbor. Subsequently, the backtracking process reaches the root node $(6, 5)$. The distance from the given query point Q_2 to the splitting line $x = 6$ is 4, which is larger than the current nearest distance. As a result, node $(6, 5)$ and all the nodes on its right are pruned from the distance computation and the backtracking process is terminated.

For a balanced KD-tree, the order of steps of inserting



(a) Query point $Q_1 = (2.5, 2)$



(b) Query point $Q_2 = (2, 5)$

Fig. 2. Examples of nearest neighbor query in KD-tree

a node into the tree is the height of tree, which is the logarithm of the node number of the current tree in general. Thus, to construct a KD-tree with n nodes, the corresponding overhead order is $O(n \log n)$. To find the nearest neighbor of a given query point, only a few nodes in the KD-tree may be visited during the backtracking process, as demonstrated in the above examples. Therefore, the computational overhead of the nearest neighbor query is usually in the order of a small number of nodes. At worst, the overhead of each time of querying is in the order of $O(n)$.

III. The Proposed KDFC-ART Approach Based on KD-Trees

A. The Framework

A major criticism of the original FSCS-ART algorithm lies in the relatively expensive generation of test cases. In this paper, we adopt the KD-tree structure to alleviate this problem in the FSCS-ART algorithm. The improvement is mainly due to a novel strategy to store the already executed test cases in a KD-tree. In RT or its variants, test cases are incrementally generated. Thus, the tree should be dynamically updated to store the incrementally selected test cases. Consequently, the nearest neighbor query for a candidate should be conducted on the most updated KD-tree.

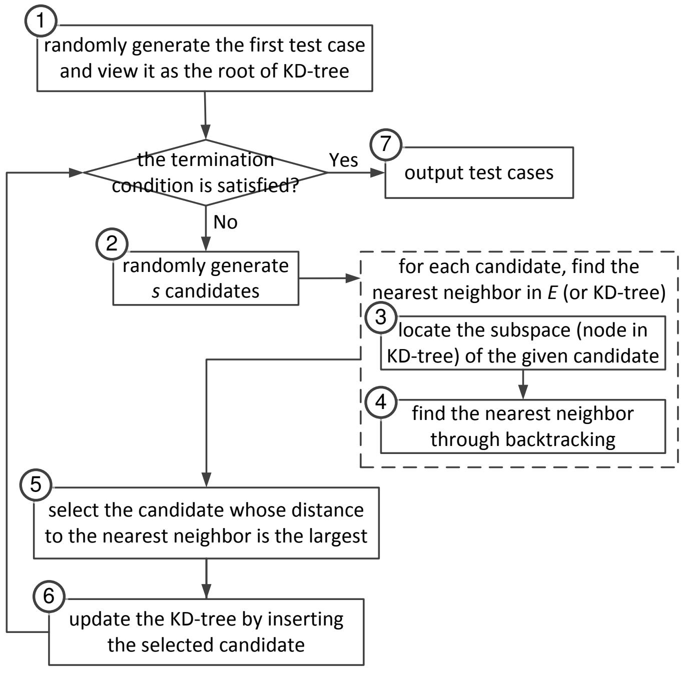

1
randomly generate the first test case and view it as the root of KD-tree
the termination condition is satisfied?
Yes
No
7
output test cases
2
randomly generate s candidates
for each candidate, find the nearest neighbor in E (or KD-tree)
3
locate the subspace (node in KD-tree) of the given candidate
4
find the nearest neighbor through backtracking
5
select the candidate whose distance to the nearest neighbor is the largest
6
update the KD-tree by inserting the selected candidate

Fig. 3. Flowchart of KDFC-ART

The whole process is illustrated in Figure 3. Initially, the first test case is randomly generated and considered as the root of KD-tree (step 1). The subsequent test cases are generated one by one through an iterative procedure. In each round of iterations, a fixed-size candidate set with s candidates is randomly generated from the input space (step 2). Then, for each candidate, the algorithm finds its nearest neighbor from the executed test set E (or the current KD-tree). Specifically, the action of finding the nearest neighbor can be divided into two steps: In step 3, our method locates the subspace (i.e., potential parent node in the tree) containing the current candidate downward from the root node according to its coordinates. Then, the corresponding potential parent node is regarded as a temporary neighbor nearest to the candidate. Subsequently, step 4 further queries the nearest neighbor of the candidate by the backward traversal from the potential parent node. If the backtracking is not bounded, the queried nearest neighbor is the actual one, otherwise not necessarily the actual nearest neighbor. After finding out the nearest neighbors of all s candidates, the candidate with the farthest nearest neighbor distance is selected as the next test case (step 5). Once the next test case (i.e., the selected candidate) is determined, it should be executed and inserted into the storage structure of the executed test set E, i.e. KD-tree. This action is shown in step 6 of Figure 3. Furthermore, it is to check the termination condition of test case generation. In general, the condition could be set as follows: (1) at least one fault in the program under test has been detected or (2) the number of executed test cases reaches a predefined ceiling. When the termination condition is met, the test case generation process will stop and report the test results (step 7).

It is not hard to find that the following three steps in the test case generation process involve relatively heavy computational overheads: Both steps 3 and 6 need to find a position for a query point which is actually an element of the candidate set, in the tree through the binary search. The query cost is in the order of the height of KD-tree. For the balanced tree with n nodes, the cost is about $\log n$ for each query. If the tree is unbalanced, in the worst case, the cost of each query may be proportional to number of nodes in the tree, that is, $O(n)$. Therefore, in order to reduce the query cost in these two steps, KD-tree should be kept as balanced as possible. Another computationally expensive step is the backtracking for finding the nearest neighbor of each candidate. Although the number of nodes traversed in practice is usually quite small, the worst case may need to traverse nearly all nodes. Hence, it is better to have a strategy to control the number of nodes in backtracking in order to set an upper bound on the time to be spent on searching for the nearest neighbor.

Based on the above analysis, we have modified the original FSCS-ART to form the KDFC-ART algorithms in the following three directions. (1) Improve FSCS-ART using the KD-tree structure to store the generated (or executed) test cases so as to speed up the nearest neighbor query for candidates. (2) Improve FSCS-ART by using a heuristic strategy to construct a KD-tree with better balance. (3) Further improve the efficiency of FSCS-ART by limiting the number of backtrackings. Of course, the nearest neighbor found in this way may only be approximately nearest.

B. KDFC-ART Based on Round-Robin Strategy

Normally, KD-tree is constructed for a static data set of a given size. Since the data set is known beforehand, the partitioning in the space (i.e., the splitting of the nodes of in the tree) can be conducted by referring to the known variance and the known median of the points in each dimension. The dimension with the maximum variance of data points' coordinates will be split using the hyperplane at the median point. In RT and its variants, test cases in the input space can be generated in parallel with the testing process. As a result, the variance and median change continuously. We need to look for alternative strategies to construct a KD-tree incrementally to store test cases. This section presents the first option.

In order to implement the dynamic updating of a KD-tree, we first propose a naive strategy as the priority of splitting. Take the 2-dimensional input space in Figure 4 as an example. For the root node at the first level, we select dimension X to split. Then, we choose dimension Y to split for the children. It is followed by splitting dimension X again for all the grandchildren, followed by splitting dimension Y again for all the great-grandchildren, and so on.

Specifically, for the dynamic generation of test cases, the splitting hyperplane of the new-coming test case can be determined as follows: Once a candidate is identified as the next test case tc, we need to determine the smallest subspace that contains it. This subspace is represented by a potential parent node in the KD-tree. Then, tc should be added as a left or right child node of the potential parent. Suppose the

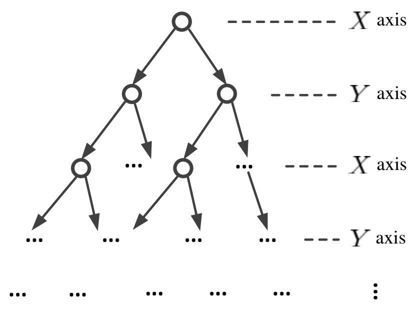


Fig. 4. Example of applying the round-robin strategy to split the input space ($d = 2$)

subspace represented by the potential parent should be split with respect to the ith dimension ($i \in \{1, 2, \ldots, d\}$). Then, the new subspace represented by tc should be split with respect to the $(i+1)$th dimension if $i < d$. Of course, if $i = d$, it should be split with respect to the 1st dimension.

Based on the above analysis, we can enhance FSCS-ART using a round-robin KD-tree strategy. Its pseudocode is listed in the Naive-KDFC algorithm. In addition to the input space information and the number of test cases, a termination condition is also accepted as an input. In general, the termination condition can be set to a maximum number of test cases generated, or when the first failure of the program has been found. In the algorithm, the generated test cases are stored in *TCset* and the corresponding structure is captured in *KDtree*. The former is used for the final output while the latter is to support fast queries of nearest neighbors. Line 1 initializes the KD-tree and the set of test cases. Lines 2–3 randomly generate the first test case and insert it into the KD-tree as the root node. Lines 4–7 split the input space with respect to the first coordinate of the test case. We then enter an iteration of processes for subsequence test cases as long as the termination condition has not been reached (line 8). First, randomly generate s different candidates (line 9), and then find the one farthest away from its nearest neighbor as the next test case tc (lines 10–15).

In particular, given a candidate, there are two steps to find the nearest neighbor in *KDtree*. The first step is to locate the potential parent node representing the smallest subspace containing the candidate (line 11). The second step is to incrementally update the temporary nearest neighbor through searching and backtracking until the true nearest neighbor is found (line 12). (In the worst case, this second process may cover nearly the whole input space, or nearly all the nodes in the KD-tree, before finding the nearest neighbor.) After the nearest neighbors of all candidates are found, the candidate with the farthest distance from its nearest neighbor is taken as the next test case tc (lines 14–15).

Next, locate the potential parent for node tc (line 16) and insert tc as a left or right child node (lines 17–22). Meanwhile, update the round-robin counter of splitting dimensions (lines 23–27). For the recently inserted node tc, set the splitting dimension, the splitting hyperplane, and perform the splitting accordingly (lines 28–30). Finally, insert tc into *TCset* for output purpose (line 31).

Algorithm 1. Naive-KDFC

```
Inputs: (a) The size $s$ of every candidate set, where $s > 1$;
        (b) The number of dimensions $d$ in the program input space;
        (c) The termination condition for the test case generation process;
Output: The set of test cases $TCset$;
 1. Set $KDtree = \{\}$ and $TCset = \{\}$;
 2. Randomly select a test case $tc$ from the input space;
 3. Insert $tc$ into $KDtree$ as the root node;
 4. Set $m = 1$;
 5. Set the splitting dimension of node $tc$ = $m$;
 6. Set the splitting hyperplane of this dimension at the value of the
    $m$th coordinate of $tc$;
 7. Split the input space according to this hyperplane;
 8. while (termination condition is not satisfied) do
 9.     Randomly select $s$ candidates $cand_1, cand_2, \ldots, cand_s$ from
        the input space;
10.     for each candidate $cand_k$ ($k = 1, 2, \ldots, s$) do
11.         Find the smallest subspace containing $cand_k$ and hence find
            the potential parent of node $cand_k$ in $KDtree$;
12.         Starting from the potential parent of node $cand_k$, find the
            nearest neighbor of $cand_k$ through searching and backtracking in $KDtree$;
13.     end for
14.     Find $cand_{\text{best}}$ from $cand_1, cand_2, \ldots, cand_s$ having the maximum
        distance from its nearest neighbor;
15.     Set $tc$ = $cand_{\text{best}}$;
        // Note that the smallest subspace containing $tc$ is the same
        as the smallest subspace containing $cand_{\text{best}}$
16.     Set node $parent_{tc}$ = the potential parent of node $cand_{\text{best}}$;
17.     Set $m$ = the splitting dimension of node $parent_{tc}$;
18.     if the value of the $m$th coordinate of $tc \leq$ the value of the
        $m$th coordinate of $parent_{tc}$ then
19.         Insert $tc$ into $KDtree$ as a left child of node $parent_{tc}$;
20.     else
21.         Insert $tc$ into $KDtree$ as a right child of node $parent_{tc}$;
22.     end if
23.     if $m < d$ then
24.         Set $m = m + 1$;
25.     else
26.         Set $m = 1$;
27.     end if
28.     Set the splitting dimension of node $tc$ = $m$;
29.     Set the splitting hyperplane of this dimension at the value of
        the $m$th coordinate of $tc$;
30.     Split the subspace according to this hyperplane;
31.     Insert $tc$ into $TCset$;
32. end while
33. return $TCset$;
```

Briefly speaking, the Naive-KDFC algorithm takes turns to use each dimension (from 1 to d) as a hyperplane criterion for splitting. This method can be easily implemented, but cannot guarantee the delivery of an unbiased KD-tree. In general, if the tree is heavily skewed, it will incur expensive node insertion, node query, and backtracking to find the nearest neighbor. It is therefore necessary to further improve the construction process of the KD-tree.

C. KDFC-ART Based on Semi-Balanced Strategy

As already mentioned, the naive algorithm Naive-KDFC may lead to a skewed KD-tree, which will in turn incur expensive node localizations and nearest neighbor queries. In order to improve the balance of the target KD-tree, let us look deeper into issue.

As illustrated in Figure 5, suppose node tc_j lies in the smallest subspace shown by the square region. It is the turn to use the X-axis to split the subspace. Here, the splitting hyperplane is represented by $x = x_j$. Accordingly, the whole area is divided into two parts: the left subregion L in which the X-coordinates of all the points $\leq x_j$, and the right subregion R where the X-coordinates of all the points $> x_j$. With regard to the corresponding tree, the points in the left subregion L belong to the left subtree of node n_j, and all the points in the right subregion R are in the right subtree. Intuitively speaking, the number of nodes in the left (right) subtree is proportional to the area of the corresponding left (right) subregion. Thus, if we want to make the KD-tree unskewed, the ranges of the two subtrees after splitting need to be as equal as possible. Obviously, the naive method cannot ensure an unskewed KD-tree. As a consequence, it is necessary to apply another strategy to split the input space.

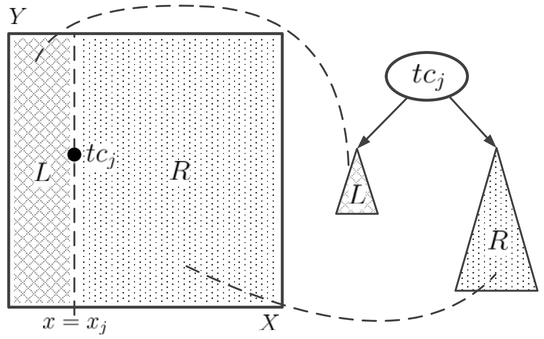


Fig. 5. Example of unbalanced splitting of input (sub)space

Given a d-dimensional Euclidean space $\boldsymbol{X} = X_1 \times X_2 \times \cdots \times X_d$, the *range* ℓ_i for dimension X_i ($i \in \{1, 2, \ldots, d\}$) is defined as the length between the maximum and minimum values in X_i. For example, given a 2-dimensional space $[1, 10] \times [-5, 5]$, the ranges for the first and second dimensions are 9 and 10, respectively.

Let $X_1, X_2, \ldots, X_d$ be the dimensions of the input space. Suppose that we would like to add a test case to an existing test set (represented by a KD-tree). We may recall that the range ℓ_i of dimension X_i of the subspace containing the test case is the distance between the minimum and maximum possible values of the X_i components of the points in the subspace. The test case can divide every dimension into two parts. We denote the resulting ranges of these two parts by ℓ_{i1} and ℓ_{i2}, respectively, where $\ell_{i1} + \ell_{i2} = \ell_i$. Then, we define the *spread* of dimension X_i as follows:

$$spread(i) = \ell_i \; gini(\ell_{i1}, \ell_{i2}), \tag{2}$$

$$gini(\ell_{i1}, \ell_{i2}) = 1 - \left(\frac{\ell_{i1}}{\ell_i}\right)^2 - \left(\frac{\ell_{i2}}{\ell_i}\right)^2 \tag{3}$$

where $gini(\ell_{i1}, \ell_{i2})$ is the *Gini coefficient* [28] of the ranges ℓ_{i1} and ℓ_{i2} of the two parts $i1$ and $i2$ ($i \in \{1, 2, \ldots, d\}$), named after Gini [29] in 1912. Out of the d dimensions, the SemiBal-KDFC algorithm selects dimension X_i with the largest value of $spread(i)$ to split.

Let us consider two examples in 2-dimensional input spaces to explain the priority of splitting. In Figure 6(a), the ranges of the two dimensions X and Y are equal, but $gini(\ell_{x1}, \ell_{x2}) > gini(\ell_{y1}, \ell_{y2})$ according to (3), giving $spread(X) > spread(Y)$. As a result, we select dimension X to split. In Figure 6(b), the range of dimension Y is greater than that of dimension X. Although the Gini coefficient of dimension X is the same as that of dimension Y ($gini(\ell_{x1}, \ell_{x2}) = gini(\ell_{y1}, \ell_{y2})$), the spread of dimension Y is relatively higher ($spread(Y) > spread(X)$). Thus, we choose to split dimension Y in this case.

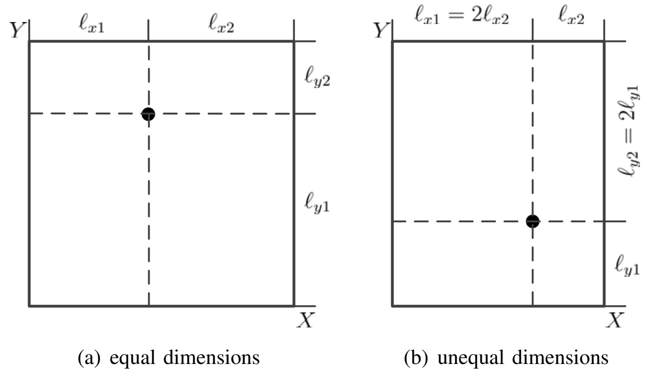


Fig. 6. Examples of selecting the splitting dimension

Based on these two examples, it is easy to see that our proposed splitting criterion mainly emphasizes the following two aspects: (1) The dimension on which the splitting is relatively more balanced will be a likely choice for splitting. (2) The dimension with a greater range will also be a likely choice for splitting. Since the test cases are randomly selected from input space, the space or subspace is divided more balanced usually means that the numbers of nodes in the corresponding left and right subtrees are more likely to be the same. Intuitively, the preferred division on the relatively long dimension usually can reduce the range of backtracking traversal. Accordingly, the distance computation cost can be reduced to some extent.

Here, we refer to the enhanced version of FSCS-ART based on a more balanced KD-tree as the SemiBal-KDFC algorithm. Compared with the Naive-KDFC algorithm, SemiBal-KDFC can build an improved KD-tree by applying the splitting strategy introduced earlier in this section. In essence, we replace the round-robin strategy of Naive-KDFC by the following concept in the SemiBal-KDFC algorithm: *Identify the next dimension to split as the one having the maximum spread according to equations (2) and (3).* More details are listed in the pseudocode below. Since a more balanced KD-tree can be produced by the revised splitting strategy, SemiBal-KDFC can reduce the computation time of node insertion and nearest neighbor query.

D. KDFC-ART Based on Semi-Balanced and Restricted Backtracking Strategy

The efficiency of FSCS-ART method is mainly dominated by the distance computation of finding the nearest neighbor for each candidate. With the KD-tree representation of test cases, the search process for a candidate's nearest neighbor

Algorithm 2. SemiBal-KDFC

```
Inputs: (a) The size $s$ of every candidate set, where $s > 1$;
    (b) The number of dimensions $d$ in the program input space and
    the minimum and maximum possible values of each dimension;
    (c) The termination condition for the test case generation process;
Output: The set of test cases $TCset$;
 1. Set $KDtree = \{\}$ and $TCset = \{\}$;
 2. Randomly select a test case $tc$ from the input space;
 3. Insert $tc$ into $KDtree$ as the root node;
 4. Set $m$ = the next dimension to split according to equations (2) and (3);
 5. Set the splitting dimension of node $tc$ = $m$;
 6. Set the splitting hyperplane of this dimension at the value of the $m$th coordinate of $tc$;
 7. Split the input space according to this hyperplane;
 8. while (termination condition is not satisfied) do
 9.     Randomly select $s$ candidates $cand_1, cand_2, \ldots, cand_s$ from the input space;
10.     for each candidate $cand_k$ $(k = 1, 2, \ldots, s)$ do
11.         Find the smallest subspace containing $cand_k$ and hence find the potential parent of node $cand_k$ in $KDtree$;
12.         Starting from the potential parent of node $cand_k$, find the nearest neighbor of $cand_k$ through searching and backtracking in $KDtree$;
13.     end for
14.     Find $cand_{\text{best}}$ from $cand_1, cand_2, \ldots, cand_s$ having the maximum distance from its nearest neighbor;
15.     Set $tc$ = $cand_{\text{best}}$;
        // Note that the smallest subspace containing $tc$ is the same as the smallest subspace containing $cand_{\text{best}}$
16.     Set node $parent_{tc}$ = the potential parent of node $cand_{\text{best}}$;
17.     Set $m$ = the splitting dimension of node $parent_{tc}$;
18.     if the value of the $m$th coordinate of $tc \leq$ the value of the $m$th coordinate of $parent_{tc}$ then
19.         Insert $tc$ into $KDtree$ as a left child of node $parent_{tc}$;
20.     else
21.         Insert $tc$ into $KDtree$ as a right child of node $parent_{tc}$;
22.     end if
23.     Set $m$ = the next dimension to split according to equations (2) and (3);
24.     Set the splitting dimension of node $tc$ = $m$;
25.     Set the splitting hyperplane of this dimension at the value of the $m$th coordinate of $tc$;
26.     Split the subspace according to this hyperplane;
27.     Insert $tc$ into $TCset$;
28. end while
29. return $TCset$;
```

can be described as below: Once a candidate is located into a subspace (i.e. the potential parent node of the candidate in KD-tree), the test case represented by the potential parent node is viewed as the temporary nearest neighbor of the candidate. Then, a backtracking process is used to find the actual nearest neighbor among the test set (i.e. the nodes in the tree). Here, the backtracking is conducted along with a path which starts from the located potential parent node to the root of KD-tree. For each node in the backtracking path, the distance from the candidate to the splitting hyperplane, which is represented by the equation attached to the node (such as $x = 7$ in Figure 1(b)), is calculated first. If this distance is less than the current nearest distance, the other branch of the node with respect to the backtracking branch should be further searched. Otherwise, the backtracking process will go backward to the node in the upper level.

Since the nodes involved with backtracking search can be pruned according to the distance from the candidate to the splitting hyperplane, the distance computation overhead in Naive-KDFC and SemiBal-KDFC algorithms can obviously be reduced. The improvement in efficiency is quite significant for low-dimensional input spaces. In general, the number of traversed nodes during the backtracking process is usually quite small compared with the size of the test set. Specifically, finding the nearest neighbor in a balanced KD-tree with randomly distributed points takes $O(\log n)$ time on average [26]. However, the backtracking search may traverse unexpectedly many nodes in the worst scenario. In particular, in the case of high-dimensional input spaces, the curse of dimensionality causes the algorithm to visit many more branches than in low-dimensional spaces. Thus, the test cases involved in the backtracking and distance computation for each candidate may be a large portion of the executed test cases. For $d = 10$, for instance, it has to query 97 nodes on average in order to find the nearest neighbor from a KD-tree with 100 nodes. For trees with 1000 and 5000 nodes, the average numbers of the queried nodes are 804 and 2227, respectively. Based on the above analysis, it may be cost-effective if we can limit the number of nodes to be queried in the backtracking search. We can find the following clues based on experiments similar to the above example: (1) A tree with more branches has more chance of pruning in backtracking search; (2) KDFC-ART algorithms are more favorable (in terms of efficiency) than FSCS-ART when the failure rate is low but not when it is high.

In our method, an upper bound is set to limit the number of backtrackings as follows:

$$bound(j+1) = \frac{1}{2}\big(d + \frac{1}{d}\big)^2 \log j \qquad (4)$$

where d is the dimension of the input space of the program and j is the ordinal number of the test case. That is to say, to generate the $(j+1)$th test case, the number of nodes for distance computation in the backtracking search should be limited to less than $bound(j+1)$. It is important to note that the distance computation mentioned here includes both the distance from the candidate to a splitting hyperplane and the distance from the candidate to an executed test case (that is, a node in the tree).

According to the above ideas, we can further improve the SemiBal-KDFC algorithm using limited backtracking. We will refer to this algorithm as LimBal-KDFC. The corresponding pseudo code is listed below. Compared with Naive-KDFC algorithm, SemiBal-KDFC algorithm attempts to strike a better balance between two split subspaces. This strategy is also employed in LimBal-KDFC algorithm. Furthermore, in the LimBal-KDFC algorithm, only a limited number of backtrackings can be applied (see line 12). Based on this limit, LimBal-KDFC can further reduce the distance computation time. Of course, the distance between a given candidate and its nearest neighbor is only approximate rather than precisely calculated. On the other hand, as shown in the results of the follow-up experiments, such an approximation does not induce significant deterioration of the failure-detection effectiveness.

Algorithm 3. LimBal-KDFC

```
...
12. Starting from the current potential parent of node $cand_k$,
    find the nearest neighbor of $cand_k$ through searching and
    backtracking in KDtree, where the backtracking process
    is limited by the bound in equation (4);
...
```

E. Summary of the Three KDFC-ART Algorithms

For the above three enhanced versions of FSCS-ART, we summarize their characteristics from the five aspects in Table I. For the sake of a fair choice among various dimensions during KD-tree construction, the Naive-KDFC algorithm adopts a round-robin technique, in which each dimension is successively selected for splitting when partitioning the input space. SemiBal-KDFC and LimBal-KDFC utilize a dynamic balancing strategy to adaptively select the dimension with the maximum spread of points as the next dimension to split based on equations (2) and (3). With respect to the backtracking strategy, there is no limit in the backtracking steps in both Naive-KDFC and SemiBal-KDFC. On the contrary, in the LimBal-KDFC algorithm, the backtracking steps are limited under the upper bound defined in equation (4). As a result, the overheads of distance computations between candidates and the executed test cases will be limited. Of course, this mechanism has an adverse effect that the "nearest neighbor" of a candidate test case may not be truly nearest. By contrast, Naive-KDFC and SemiBal-KDFC guarantee the identification of the actual nearest neighbor for the candidate. However, unlimited backtracking may require a large number of distance computations, especially for high-dimensional input spaces.

As stated in Section III-A, the time complexity of KD-tree-based FSCS-ART algorithms mainly involves the following two parts. The first part is the computation of locating a test case or a candidate to a node of KD-tree. Since all three algorithms adopt KD-tree structure to store the executed test cases and the test cases are randomly selected from the whole input space, the heights of the trees built by the above three algorithms are basically in the order of $\log j$, where j is the number of nodes in the KD-tree. Although SemiBal-KDFC and LimBal-KDFC can construct more balanced KD-trees than Naive-KDFC, the tree heights in three algorithms are roughly at the same order of magnitude. Thus, for a test set of n test cases, the computations of three algorithms to localize all test cases in the tree are the order of $O(n\log n)$. Similarly, the localization computations of all s candidates for generating n test cases are the order of $O(sn\log n)$.

The second part is the computation of nearest neighbor query by backtracking in KD-tree. On average, finding the nearest neighbor in the KD-tree whose nodes are randomly distributed points takes $O(\log n)$ time [26]. Therefore, to generate a test set with n test cases, the average overheads of distance computation in all three algorithms are the order of $O(sn\log n)$. However, in the worst case, the backtracking in Naive-KDFC and SemiBal-KDFC may traverse almost all nodes in the tree. Thus, the worst complexities of distance computation in Naive-KDFC and SemiBal-KDFC are the order of $O(sn^2)$. Because the backtracking in LimBal-KDFC is limited within the bound of $\frac{1}{2}(d+\frac{1}{d})^2\log j$, its worst complexity of distance computation is still the order of $O(d^2sn\log n)$.

In a word, the average time complexities of three enhanced algorithms are the order of $O(sn\log n)$. The worst time complexities of Naive-KDFC and SemiBal-KDFC are the order of $O(sn^2)$, but the worst complexity of LimBal-KDFC is the order of $O(d^2sn\log n)$, where n is number of test cases, and d and s are two constants, which are the dimension of the program's input space and the fixed size of candidate set, respectively.

IV. Simulation Studies

A. Experimental Setup and Evaluation Metrics

The aim of this research is to improve the efficiency of the popular FSCS-ART algorithm. Hence, it is very necessary to confirm whether our enhanced algorithms can reduce the test case generation time. In this paper, we use the execution time of generating test cases as an efficiency metric for comparison analysis. The *Wilcoxon rank-sum test* [30] (or equivalently the *Mann-Whitney U test* [31]) is used to verify whether there are differences in efficiency among the investigated ART algorithms. Meanwhile, *effect size* [32] is used to measure the magnitudes of the differences among ART algorithms. The effect size for the Wilcoxon rank-sum test [33] is given by the correlation coefficient

$$r=\frac{|z|}{\sqrt{n_1+n_2}} \tag{5}$$

where z is the z-value in rank-sum test results, and n_1 and n_2 represent the number of observations in two populations. According to the guidelines given by Cohen [34], a rough interpretation of effect size is that $r=0.5$ represents a large effect, $r=0.3$ represents a medium effect, and $r=0.1$ represents a small effect.

For each candidate, both Naive-KDFC and SemiBal-KDFC can find the exact nearest neighbor, which is the same as FSCS-ART because they generate the same set of test cases. As a consequence, the failure-detection capabilities of these two enhanced algorithms are identical to that of FSCS-ART. Hence, it is unnecessary to perform the comparisons of failure-detection effectiveness among Naive-KDFC, SemiBal-KDFC, and the original FSCS-ART. However, the backtracking steps are limited in LimBal-KDFC, it is necessary to carry out an in-depth analysis and comparison on its effectiveness. In this section, we mainly report the simulation results.

As described earlier, failure patterns in programs can be classified into three categories: *block patterns*, *strip patterns*, and *point patterns*. These three regular failure patterns are used in the simulation of failure-causing input regions of a program. Once a test case is selected inside such a region, a program failure is deemed to have occurred. To simulate a block pattern in the experiments, we used a hypercube having sides of equal lengths (such as a square or a cube in 2- or 3-dimensional spaces) as the failure region. Only one such region was randomly placed in the input space. To simulate a strip pattern, a strip at any angle was treated as the failure region. Since the strips near the corners of the input space will be "fat" rather than real strips, these cases were excluded

TABLE I
CHARACTERISTICS OF THE THREE KDFC-ART ALGORITHMS

Property	Naive-KDFC	SemiBal-KDFC	LimBal-KDFC
Balancing strategy	Round robin: splitting each dimension iteratively from 1 to d	Semi-balancing strategy: considering the range and Gini coefficient incrementally for each test case	Semi-balancing strategy as per SemiBal-KDFC
Backtracking strategy	No limit on number of steps	No limit on number of steps	Limited number of steps
Nearest neighbor query	Exact result, test cases the same as FSCS-ART	Exact result, test cases the same as FSCS-ART	Approximate result, test cases different from FSCS-ART
Average complexity	$O(sn\log n)$	$O(sn\log n)$	$O(sn\log n)$
Worst case complexity	$O(sn^2)$	$O(sn^2)$	$O(d^2sn\log n)$

from the strip pattern simulation. For the point pattern, 25 non-overlapping little squares or cubes were randomly generated from the input space according to the corresponding failure rate setting.

Automated test oracles are assumed as follows: In simulation studies, the simulated failure patterns act as the test oracles, and in empirical studies in the next section, the original versions of the subject programs are used as the test oracles for their mutants. In the context of software testing, the expected number of test cases to detect the first failure is usually defined as an effectiveness metric, namely, *F-measure* [13], [35]. In addition, another metric (known as *F-ratio*) for adaptive random testing can be further defined as follows: *F-ratio* $= F_{\text{ART}}/F_{\text{RT}}$, where F_{ART} and F_{RT} are the F-measures of the ART method and the RT method, respectively. The value of F_{RT} in the simulation studies can be directly computed from the failure rate using the formula $F_{RT} = 1/\theta$. Here, θ is the *failure rate* of the program, computed as the ratio of the total area (or volume) of the failure-causing regions to the total area (or volume) of the input domain in the simulation study.

The experiments were conducted using a desktop PC with i7 CPU at 3.6 GHz and 8 GB RAM running under the 64-bit Windows 7 operating system. All the algorithms were implemented in Java[4] and executed on the Eclipse platform with JDK 1.8.

In this study, our aim is to improve the efficiency performance of a special ART algorithm, namely FSCS-ART, which is the first proposed ART algorithm and is also known as one of ART methods that have the best failure-detection capability. Therefore, we focus on the comparison with FSCS-ART in the following studies.

B. Study of Naive-KDFC Algorithm

The main motivation of the project is to reduce the computation time of the original FSCS-ART algorithm. In this section, we conduct efficiency comparisons between Naive-KDFC and FSCS-ART. The study attempts to answer the following research question:

[4]The implementation of our three algorithms is available at https://github.com/maochy/kdfc-art.

RQ1. Does Naive-KDFC spend less time than FSCS-ART in generating the same number of test cases?

In the comparison analysis, for each test set size, Naive-KDFC and FSCS-ART are both repeated in 1000 trials and their execution times are recorded to obtain the mean values, effect sizes, and p-values in rank-sum tests. The results are listed in Tables II and III. For 2-dimensional input spaces (under the column $d = 2$), the test case generation time by Naive-KDFC is slightly less than that by FSCS-ART when $n \leq 200$. When the number of test cases increases to 500, the advantage of our Naive-KDFC algorithm becomes more obvious. After that, the generation time by Naive-KDFC is always markedly less than that of FSCS-ART as the number of test cases increases. For 3-dimensional input spaces (under $d = 3$), Naive-KDFC is slower than FSCS-ART when $n \leq 100$. Although the gap is quite small (only 0.08 ms), the difference is statistically significant. The underlying reason of this phenomenon is that the pruning effect for finding the nearest neighbor is not prominent when there are fewer nodes in the KD-tree, especially for high-dimensional spaces. On the other hand, Naive-KDFC requires a certain computational cost to build the KD-tree and additional cost to compute the distances from the candidates to the splitting hyperplanes. When n goes beyond 100, the test case generation time for Naive-KDFC is significantly less than that of FSCS-ART because of the large effect size and because p-value $\ll 0.05$.

In the case of $d = 5$, Naive-KDFC is significantly slower than FSCS-ART when the number of test cases n is less than 500. However, when $n \geq 500$, the test case generation time of Naive-KDFC is always less than that of the original FSCS-ART and the amount of improvement is large and statistically significant. When the $d = 10$, the range of degradation of Naive-KDFC becomes wider. Here, Naive-KDFC is efficient only when $n \geq 5000$. The above phenomena show that the efficiency of the Naive-KDFC algorithm is reduced in the case of high-dimensional input spaces, but can still be effective when the number of test cases is large.

The trends of execution times for these two algorithms with respect to the number of test cases are demonstrated in Figure 7. The computation time of the original FSCS-ART algorithm increases dramatically with the growing number of test cases. As analyzed in Section II.A, the time complexity of the FSCS-

TABLE II
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON RUNNING TIME BETWEEN NAIVE-KDFC AND FSCS-ART FOR $d = 2$ AND $d = 3$

No. of test cases n	$d = 2$				$d = 3$			
	Mean of running times (ms)		p-value	Effect size	Mean of running times (ms)		p-value	Effect size
	FSCS-ART	Naive-KDFC			FSCS-ART	Naive-KDFC		
100	0.33	0.31	0.0000	0.4574	0.40	0.48	0.0000	0.6644
200	1.07	0.67	0.0000	0.8216	1.29	1.17	0.0000	0.5632
500	6.84	2.04	0.0000	0.8658	7.84	3.64	0.0000	0.8546
1000	28.20	4.41	0.0000	0.8658	32.05	8.42	0.0000	0.8658
2000	123.54	10.48	0.0000	0.8658	148.47	20.22	0.0000	0.8658
5000	868.35	31.15	0.0000	0.8658	1007.23	61.57	0.0000	0.8658
10000	3480.89	68.68	0.0000	0.8658	4047.33	138.89	0.0000	0.8658
15000	7763.46	109.89	0.0000	0.8658	9045.80	220.81	0.0000	0.8658
20000	13575.36	150.94	0.0000	0.8658	16554.91	306.21	0.0000	0.8658

TABLE III
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON RUNNING TIME BETWEEN NAIVE-KDFC AND FSCS-ART FOR $d = 5$ AND $d = 10$

No. of test cases n	$d = 5$				$d = 10$			
	Mean of running times (ms)		p-value	Effect size	Mean of running times (ms)		p-value	Effect size
	FSCS-ART	Naive-KDFC			FSCS-ART	Naive-KDFC		
100	0.47	0.89	0.0000	0.8550	0.77	1.54	0.0000	0.8546
200	1.61	2.49	0.0000	0.8506	2.65	5.73	0.0000	0.8658
500	10.06	9.54	0.0000	0.5256	16.21	32.72	0.0000	0.8658
1000	43.71	25.22	0.0000	0.8658	66.11	114.50	0.0000	0.8658
2000	197.57	65.20	0.0000	0.8658	313.64	343.45	0.0000	0.8205
5000	1319.66	212.93	0.0000	0.8658	2257.42	1709.48	0.0000	0.8529
10000	5569.60	491.69	0.0000	0.8658	9827.84	5681.22	0.0000	0.8658
15000	12355.94	793.39	0.0000	0.8658	22178.47	11189.24	0.0000	0.8658
20000	22739.61	1119.45	0.0000	0.8658	39481.15	17340.57	0.0000	0.8658

ART algorithm is quadratic with respect to the number of test cases. Thus, we use a quadratic curve to fit the graph for the execution times of the FSCS-ART algorithm, as represented by the dashed curve in Figure 7(a). Taking $d = 2$ as an example, the relationship between time (t) and the number of test cases (n) is $t = 0.00003303 * n^2 + 0.020648 * n - 25.721$, and the sum of squared errors (SSE) of the fitted curve is 5412.6. For the other three cases, the fitted curves and the corresponding errors are shown in Figures 7(b)–(d). By contrast, for the case of low dimensions with $d \leq 5$, the computation time of the Naive-KDFC algorithm increases very slowly as the number of test cases increases. For the case of high dimensions, the execution time of Naive-KDFC observably increases as the number of test cases increase, and may be larger than that of FSCS-ART even for medium number of test cases (such as $n = 2000$ for $d = 10$). However, it still shows obvious advantage over FSCS-ART in the case of a large number of test cases. In addition, for each test case size, the variance of the running times of the Naive-KDFC algorithm is significantly smaller than that of the FSCS-ART algorithm. This means that Naive-KDFC is more stable than FSCS-ART.

Answer to RQ1: Based on the above observations, we conclude that the Naive-KDFC algorithm can reduce the computation time of the original FSCS-ART in the case of low-dimensional input spaces. It can also have an advantage in the case of high-dimensional spaces when the number of test cases is large.

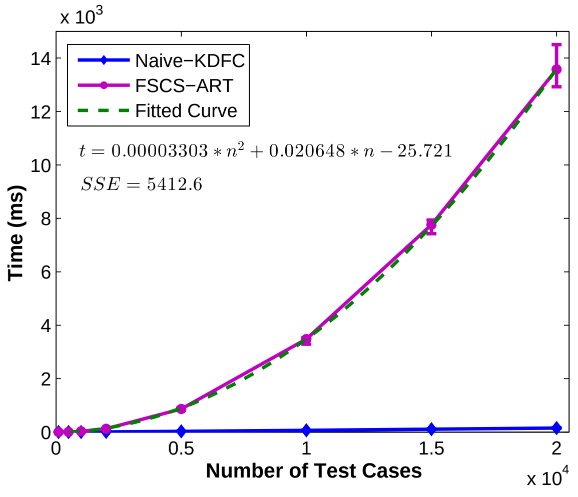
C. Study of SemiBal-KDFC Algorithm

SemiBal-KDFC is a further enhancement of Naive-KDFC by applying a more balanced splitting strategy with a view to producing a tree with shorter path lengths. In order to validate the effect of the balancing strategy, we need to investigate the following two research questions.

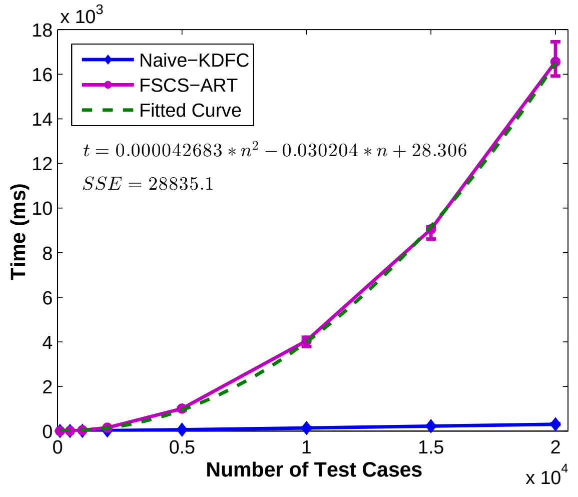
RQ2. Is the tree constructed by SemiBal-KDFC significantly more balanced than that by Naive-KDFC?

RQ3. Can SemiBal-KDFC further improve the efficiency of test case generation?

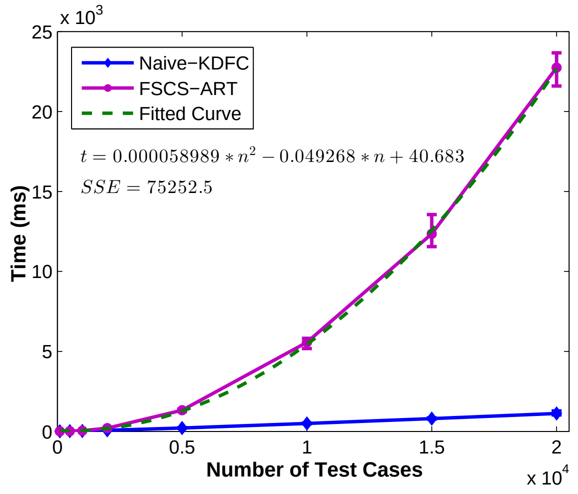
1) Investigation into the Balancing Effects of the Splitting Strategies: In order to compare the effects of the two splitting strategies in Naive-KDFC and SemiBal-KDFC respectively, we need to investigate the difference of the trees constructed by these two methods. Given a tree, its balance can be measured by the two indicators, namely, the maximum depth and the average depth of the subtrees. At each trial of experiments, the same point set is used to build KD-trees by applying algorithms Naive-KDFC and SemiBal-KDFC, respectively. In this study, the trial times were set to 1000. At the same time, the number of nodes in the tree varied from 500 to 10000, and 2-dimensional and 10-dimensional input spaces were taken into consideration. The Wilcoxon rank-sum test was also employed on the above repeated experimental results



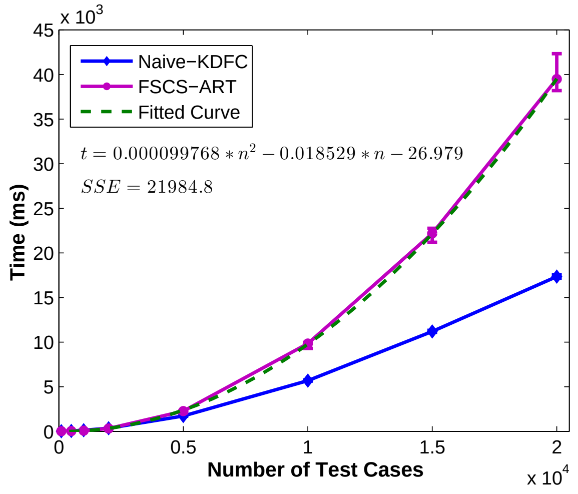
(a) $d = 2$



(b) $d = 3$



(c) $d = 5$



(d) $d = 10$

Fig. 7. Execution times for generating test cases for various test suite sizes

to evaluate the significance of the difference between Naive-KDFC and SemiBal-KDFC. The corresponding mean values (Mean), standard deviations (Std. Dev.), p-values, and effect sizes about the above two metrics are listed in Tables IV and V.

While considering the maximum depth of subtrees, in the 2-dimensional input space, the amount of improvement of SemiBal-KDFC over Naive-KDFC is sizable and the corresponding confidence level of statistical tests is always high. As shown in Table IV, the difference is about 3 when the number of nodes n in the tree is 500. Subsequently, the difference gradually increases as the number of nodes increases. When n increases to 10000, the difference in depths can reach 6. On the other hand, SemiBal-KDFC is more stable than Naive-KDFC, that is, the standard deviations of SemiBal-KDFC are always less than those of Naive-KDFC regardless of the size of the tree.

The average depths of the subtrees built by the SemiBal-KDFC algorithm are always less than those of the Naive-KDFC algorithm. The advantage of SemiBal-KDFC is also very obvious by referring to the p-values and effect sizes in Table V. The improvement also gradually increases with the increase in tree size. The standard deviations of the Naive-KDFC algorithm in the 2-dimensional and 10-dimensional input spaces have no observable difference, and are about 0.65 for all kinds of tree sizes. Based on comparison analysis on the experimental results in Table V, we can find that both mean value and standard deviation of the KD-tree average depths in the SemiBal-KDFC algorithm reduce with an increase of dimensionality. This observation indicates that SemiBal-KDFC becomes more effective and more stable as the dimension of the input space increases.

Furthermore, we also investigated the changes of above two metrics along with the increase of the dimension. The experimental results are shown in Figure 8. For the metric "maximum depth of tree", the distributions of results for the Naive-KDFC algorithm remain similar for various numbers of dimensions. However, for the SemiBal-KDFC algorithm, the distributions of maximum depths gradually decrease as the number of dimensions increases. More importantly, the distributions of maximum depths for Naive-KDFC are always larger than those for SemiBal-KDFC. For the metric of the average

TABLE IV
COMPARISONS OF MAXIMUM DEPTHS OF SUBTREES BETWEEN NAIVE-KDFC AND SEMIBAL-KDFC FOR $d = 2$ AND $d = 10$

No. of nodes n	$d = 2$						$d = 10$					
	Maximum depth				p-value	Effect size	Maximum depth				p-value	Effect size
	Naive-KDFC		SemiBal-KDFC				Naive-KDFC		SemiBal-KDFC			
	Mean	Std. Dev.	Mean	Std. Dev.			Mean	Std. Dev.	Mean	Std. Dev.		
500	19.46	1.90	16.11	1.17	0.0000	0.7755	19.37	1.86	12.78	0.65	0.0000	0.8792
1000	21.97	1.90	18.22	1.19	0.0000	0.8053	22.03	1.87	14.33	0.58	0.0000	0.8833
2000	24.80	2.01	20.35	1.28	0.0000	0.8270	24.74	1.98	15.88	0.61	0.0000	0.8818
5000	28.41	2.00	23.05	1.30	0.0000	0.8522	28.50	1.99	17.89	0.60	0.0000	0.8830
10000	31.24	2.02	25.23	1.28	0.0000	0.8637	31.18	1.98	19.41	0.61	0.0000	0.8808

TABLE V
COMPARISONS OF AVERAGE DEPTHS OF SUBTREES BETWEEN NAIVE-KDFC AND SEMIBAL-KDFC FOR $d = 2$ AND $d = 10$

No. of nodes n	$d = 2$						$d = 10$					
	Average depth				p-value	Effect size	Average depth				p-value	Effect size
	Naive-KDFC		SemiBal-KDFC				Naive-KDFC		SemiBal-KDFC			
	Mean	Std. Dev.	Mean	Std. Dev.			Mean	Std. Dev.	Mean	Std. Dev.		
500	10.64	0.67	9.48	0.32	0.0000	0.8052	10.60	0.62	8.42	0.05	0.0000	0.8658
1000	11.96	0.62	10.67	0.32	0.0000	0.8332	11.97	0.63	9.48	0.05	0.0000	0.8658
2000	13.37	0.64	11.88	0.33	0.0000	0.8484	13.37	0.66	10.54	0.04	0.0000	0.8658
5000	15.19	0.62	13.44	0.32	0.0000	0.8609	15.21	0.63	11.94	0.04	0.0000	0.8658
10000	16.59	0.64	14.69	0.34	0.0000	0.8621	16.56	0.64	13.01	0.04	0.0000	0.8658

depth of tree, the advantage of the SemiBal-KDFC algorithm is more obvious than that of the Naive-KDFC algorithm. That is, the distributions of average depths for Naive-KDFC are similar to one another while those for SemiBal-KDFC observably decrease as the number of dimensions increases.

Answer to RQ2: Based on the above results, SemiBal-KDFC can construct a more balanced KD-tree than Naive-KDFC. The statistical significance and the effect size of the difference become more profound as the number of dimensions of the input space increases.

2) Investigation into the Efficiency of SemiBal-KDFC: Intuitively, a more balanced KD-tree can help to query the nearest neighbor at a lower computational cost. Although the above experimental results have confirmed that SemiBal-KDFC can build a more balanced KD-tree than Naive-KDFC, here we still investigate whether SemiBal-KDFC can further improve the efficiency of test case generation.

As shown in Tables VI and VII, in the cases of $d = 2$, $d = 3$, and $d = 5$, the test case generation time of SemiBal-KDFC is always less than that of Naive-KDFC. When the number of test cases is small (such as when $n = 100$ or 200), the difference in computation time between the above two algorithms is small. The advantage of SemiBal-KDFC over Naive-KDFC in efficiency becomes more observable with the growing number of test cases. When $n \geq 500$, the corresponding effect size of the difference between the two algorithms is always greater than 0.4. However, for 10-dimensional input spaces, the computation time of SemiBal-KDFC is always significantly greater than that of Naive-KDFC. This phenomenon shows that, when the dimension increases to a certain extent (namely, $d \geq 9$ in our experiments), SemiBal-KDFC takes quite a bit of time to compute the spread according to equation (2). Thus, the gains derived from an enhanced balancing strategy are offset by the extra computation cost. Hence, for the case of high dimensions, the strategy in SemiBal-KDFC is not recommended.

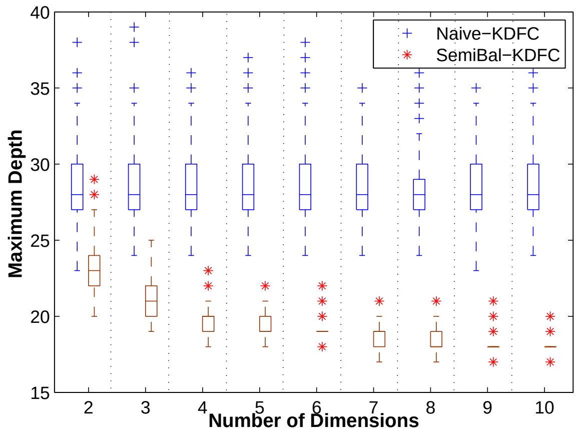
Answer to RQ3: The above experimental results confirm that SemiBal-KDFC can further improve the efficiency of test case generation in the case of low dimensions with $d \leq 8$.

D. Study of LimBal-KDFC Algorithm

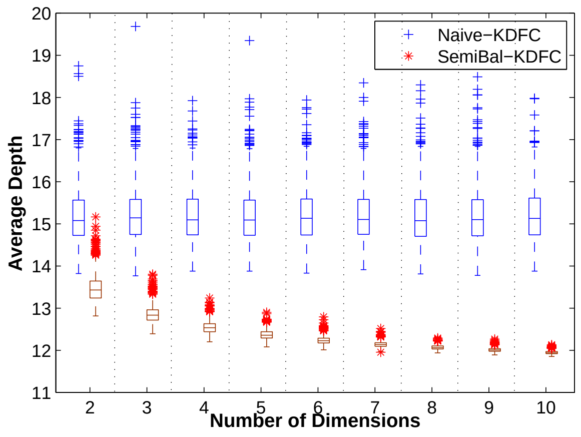
The Naive-KDFC and SemiBal-KDFC algorithms can significantly reduce the computation time in finding the nearest neighbor for a given candidate for the case of low dimensions. In most situations, they only need to traverse about $\log n$ nodes to find the nearest neighbor for a candidate in a tree with n nodes [26]. However, the worst case may result in traversing almost all the nodes in the tree. Furthermore, the performance of the nearest neighbor query in Naive-KDFC and SemiBal-KDFC will decrease as the dimension increases. In view of the above observations, a third LimBal-KDFC algorithm is proposed to limit the number of backtrackings during the nearest neighbor query.

Obviously, limited backtracking may lead to approximate rather than exact nearest neighbors for the ART candidates. Hence, we need to investigate the following two questions about the test efficiency and effectiveness of the LimBal-KDFC algorithm.

RQ4. Can LimBal-KDFC further reduce the test case generation time (especially in the case of high dimensions)?



(a) The distribution of maximum depths



(b) The distribution of average depths

Fig. 8. The distribution of KD-tree depths for different dimensions

TABLE VI
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON RUNNING TIME BETWEEN NAIVE-KDFC AND SEMIBAL-KDFC FOR GENERATING TEST CASES FOR $d = 2$ AND $d = 3$

No. of test cases n	$d = 2$				$d = 3$			
	Mean of running times (ms)		p-value	Effect size	Mean of running times (ms)		p-value	Effect size
	Naive-KDFC	SemiBal-KDFC			Naive-KDFC	SemiBal-KDFC		
100	0.31	0.29	0.0000	0.2443	0.48	0.46	0.0000	0.1553
200	0.67	0.63	0.0000	0.3460	1.17	1.07	0.0000	0.3401
500	2.04	1.83	0.0000	0.6037	3.64	3.02	0.0000	0.7361
1000	4.41	3.98	0.0000	0.6659	8.42	6.92	0.0000	0.8267
2000	10.48	9.06	0.0000	0.8203	20.22	16.43	0.0000	0.8264
5000	31.15	26.81	0.0000	0.6587	61.57	49.34	0.0000	0.8131
10000	68.68	60.38	0.0000	0.4474	138.89	109.30	0.0000	0.7965
15000	109.89	97.43	0.0000	0.4290	220.81	175.73	0.0000	0.8008
20000	150.94	133.22	0.0000	0.4581	306.21	243.72	0.0000	0.7876

RQ5. Does limited backtracking in LimBal-KDFC result in significant degradation of the failure-detection capability?

1) Investigation into the Efficiency of LimBal-KDFC: We first compare the efficiencies between LimBal-KDFC and SemiBal-KDFC. The comparison results are shown in Table VIII. For the case of $d = 2$, the test case generation time by the SemiBal-KDFC algorithm is slightly less than that by LimBal-KDFC in most cases, but the effect sizes of the differences are very small in most cases. On the contrary, when $n \geq 10000$, the time for LimBal-KDFC is reduced by about 1% in relation to the time for SemiBal-KDFC, but again the difference is not significant. For the case of $d = 3$, the mean values of computation times of these two algorithm are nearly identical when $n \leq 1000$. Although sometimes the p-value of the rank-sum test is less than 0.05, usually the effect size is very small. When the number of test cases exceeds 1000, the time for LimBal-KDFC is less than that for SemiBal-KDFC, but the effect sizes of the differences between the two algorithms are still very small in most cases.

Moreover, we further investigated the experimental results in the cases of $d = 5$ and $d = 10$. As shown in Table IX, the computation time of LimBal-KDFC algorithm is almost always less than that of SemiBal-KDFC algorithm. At the same time, the difference between the two algorithms gradually enlarges as the number of test cases increases. When the number of test cases is small (namely, when $n \leq 200$ in $d = 5$ and $n \leq 500$ in $d = 10$), the effect size of the difference is small. The difference becomes sizable when n is large. Furthermore, the difference observably increases with the increase of the number of dimensions.

Let us also compare LimBal-KDFC with FSCS-ART and Naive-KDFC (in Table III) for the cases of $d = 5$ and $d = 10$. For $d = 5$, LimBal-KDFC algorithm is more efficient than both FSCS-ART and Naive-KDFC when $n \geq 500$. The advantage becomes more notable as the number of test cases increases. For $d = 10$, the advantage of LimBal-KDFC in computation time is shown only when $n \geq 2000$. However, for high dimensions with a large number of test cases, LimBal-KDFC can markedly reduce the computation time.

In addition, we also investigated the trend of the difference between LimBal-KDFC and FSCS-ART as the number of dimensions increases. As shown in Figure 9(a), when the number of test cases $n = 500$, LimBal-KDFC is more efficient than FSCS-ART for low dimensions (namely, when $d \leq 5$). When $n = 1000$ (Figure 9(b)), the demarcation point of d with respect to the dominance relation of LimBal-KDFC and

TABLE VII
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON RUNNING TIME BETWEEN NAIVE-KDFC AND SEMIBAL-KDFC FOR GENERATING TEST CASES FOR $d = 5$ AND $d = 10$

No. of test cases n	$d = 5$				$d = 10$			
	Mean of running times (ms)		p-value	Effect size	Mean of running times (ms)		p-value	Effect size
	Naive-KDFC	SemiBal-KDFC			Naive-KDFC	SemiBal-KDFC		
100	0.89	0.87	0.6479	0.0102	1.54	1.63	0.0000	0.2760
200	2.49	2.38	0.0000	0.3350	5.73	6.27	0.0000	0.7143
500	9.54	8.37	0.0000	0.7850	32.72	38.00	0.0000	0.8324
1000	25.22	21.21	0.0000	0.8431	114.50	144.10	0.0000	0.8633
2000	65.20	53.86	0.0000	0.6715	343.45	495.19	0.0000	0.8658
5000	212.93	173.10	0.0000	0.7946	1709.48	2487.87	0.0000	0.8658
10000	491.69	407.53	0.0000	0.7975	5681.22	7597.75	0.0000	0.8658
15000	793.39	664.39	0.0000	0.8254	11189.24	14134.76	0.0000	0.8658
20000	1119.45	924.01	0.0000	0.3251	17340.57	24187.78	0.0000	0.8658

TABLE VIII
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON RUNNING TIME BETWEEN SEMIBAL-KDFC AND LIMBAL-KDFC FOR GENERATING TEST CASES FOR $d = 2$ AND $d = 3$

No. of test cases n	$d = 2$				$d = 3$			
	Mean of running times (ms)		p-value	Effect size	Mean of running times (ms)		p-value	Effect size
	SemiBal-KDFC	LimBal-KDFC			SemiBal-KDFC	LimBal-KDFC		
100	0.29	0.29	0.4162	0.0182	0.46	0.48	0.4249	0.0178
200	0.63	0.66	0.0019	0.0695	1.07	1.09	0.0000	0.1625
500	1.83	1.85	0.0000	0.1280	3.02	3.03	0.0010	0.0738
1000	3.98	4.04	0.0147	0.0546	6.92	6.90	0.0150	0.0544
2000	9.06	9.11	0.0074	0.0599	16.43	16.14	0.0000	0.2123
5000	26.81	27.18	0.0000	0.1360	49.34	49.15	0.0688	0.0407
10000	60.38	59.35	0.0000	0.1168	109.30	108.82	0.3216	0.0222
15000	97.43	96.81	0.3219	0.0222	175.73	173.31	0.0007	0.0756
20000	133.77	132.98	0.0580	0.0424	243.72	242.33	0.1425	0.0328

FSCS-ART increases to 7. In other words, for $d \leq 7$, LimBal-KDFC outperforms FSCS-ART in terms of efficiency whereas for $d > 7$, the dominance relation is reversed. When $n = 2000$ (Figure 9(c)), LimBal-KDFC is more efficient than FSCS-ART for all the cases of $d \leq 10$. Furthermore, when $n = 5000$ (Figure 9(d)), the computation time of LimBal-KDFC is less than half of FSCS-ART even for the case of $d = 10$.

Answer to RQ4: The test case generation time of LimBal-KDFC is comparable to that of SemiBal-KDFC in the case of low dimensions (with $d \leq 4$), but is observably less than that of SemiBal-KDFC for higher dimensions. Compared with the original FSCS-ART, LimBal-KDFC is more efficient in the case of low dimensions and the case of high dimensions with a large test set.

2) Investigation into the Failure-Detection Effectiveness of LimBal-KDFC: As mentioned earlier, both Naive-KDFC and SemiBal-KDFC find the exact nearest neighbor of a given candidate. Hence, the test cases generated by these two algorithms are the same as those from the original FSCS-ART algorithm. As a result, the failure-detection effectiveness of the above three algorithms are identical. By contrast, since limited backtracking is applied in LimBal-KDFC, the nearest neighbor found by LimBal-KDFC for the candidate is an approximate one, and hence LimBal-KDFC has a different set of test case as compared with the other algorithms. For this reason, it is necessary to validate whether the approximate treatment in LimBal-KDFC has a negative effect on its failure-detection capability.

Here, the effects of FSCS-ART and LimBal-KDFC for detecting failures in three kinds of patterns are deeply studied. The experiment for each kind of failure regions was repeated in 10000 trials. In each trial, the failure region was randomly placed in the input space, and was used for FSCS-ART and LimBal-KDFC, respectively. In the experiment, we also adopted the Wilcoxon rank-sum test to investigate the difference in F-ratios between these two algorithms.

In the case of $d = 2$, the F-ratio of LimBal-KDFC is always similar to that of FSCS-ART in all kinds of failure patterns and failure rates (see Table X). The p-values of the rank-sum tests in all cases are far greater than 0.05, and the effect sizes are always very small (namely, ≤ 0.01). The results show that LimBal-KDFC has no significant deterioration to FSCS-ART in failure-detection capability. Meanwhile, the change trend of the F-ratio of LimBal-KDFC for the change of failure rate is also very consistent with that of FSCS-ART. In the other two cases of $d = 3$ and $d = 4$, the p-values are also greater than 0.05 and the effect sizes are always very small, that is, LimBal-KDFC still has no significant deterioration of failure-detection capability when compared with FSCS-ART. Besides, the change trends of the F-ratios of these two algorithms

TABLE IX
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON RUNNING TIME BETWEEN SEMIBAL-KDFC AND LIMBAL-KDFC FOR GENERATING TEST CASES FOR $d = 5$ AND $d = 10$

No. of test cases n	$d = 5$				$d = 10$			
	Mean of running times (ms)		p-value	Effect size	Mean of running times (ms)		p-value	Effect size
	SemiBal-KDFC	LimBal-KDFC			SemiBal-KDFC	LimBal-KDFC		
100	0.87	0.89	0.0000	0.1079	1.63	1.62	0.0176	0.0531
200	2.38	2.37	0.0000	0.1421	6.27	6.23	0.0000	0.1774
500	8.37	8.11	0.0000	0.3780	38.00	37.80	0.0207	0.0517
1000	21.21	19.98	0.0000	0.6213	144.10	119.15	0.0000	0.8591
2000	53.86	49.97	0.0000	0.3942	495.19	289.07	0.0000	0.8658
5000	173.10	160.48	0.0000	0.5322	2487.87	922.05	0.0000	0.8658
10000	407.53	374.03	0.0000	0.5760	7597.75	2151.56	0.0000	0.8658
15000	664.39	616.48	0.0000	0.6022	14134.76	3603.84	0.0000	0.8658
20000	924.01	864.85	0.0000	0.6473	24187.78	5679.24	0.0000	0.8658

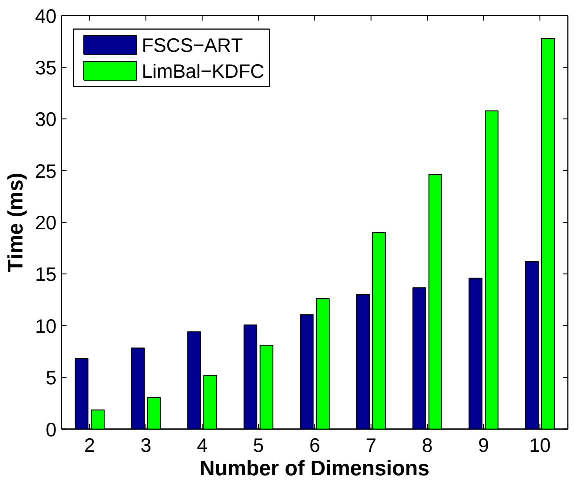
TABLE X
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON *F-ratios* BETWEEN LIMBAL-KDFC AND FSCS-ART

Dim.	Failure Rate θ	Block Patterns				Strip Patterns				Point Patterns			
		F-ratio (%)		p-value	Effect size	*F-ratio* (%)		p-value	Effect size	*F-ratio* (%)		p-value	Effect size
		FSCS-ART	LimBal-KDFC			FSCS-ART	LimBal-KDFC			FSCS-ART	LimBal-KDFC		
$d = 2$	0.0100	67.77	67.19	0.7565	0.0022	92.71	92.12	0.2221	0.0086	98.83	99.05	0.5850	0.0039
	0.0050	65.96	66.32	0.7168	0.0026	94.49	93.85	0.7124	0.0026	98.19	98.76	0.7345	0.0024
	0.0020	63.85	64.45	0.3313	0.0069	96.61	95.32	0.2989	0.0073	98.45	98.64	0.9718	0.0002
	0.0010	63.00	63.39	0.5514	0.0042	96.17	97.45	0.3710	0.0063	97.76	96.56	0.2161	0.0087
	0.0005	63.79	63.88	0.8052	0.0017	97.98	97.06	0.6274	0.0034	96.22	95.93	0.4775	0.0050
	0.0002	62.81	62.09	0.2900	0.0075	98.60	97.40	0.1587	0.0100	95.03	94.85	0.7648	0.0021
	0.0001	62.12	62.30	0.9955	0.0000	97.71	98.43	0.9529	0.0004	95.73	96.14	0.9964	0.0000
$d = 3$	0.0100	83.60	84.31	0.9563	0.0004	97.06	97.53	0.4008	0.0059	110.75	110.36	0.3296	0.0069
	0.0050	80.41	79.76	0.4820	0.0050	98.61	99.29	0.5445	0.0043	108.37	108.46	0.6764	0.0030
	0.0020	77.53	77.04	0.8420	0.0014	98.52	98.00	0.8957	0.0009	105.76	105.14	0.3684	0.0064
	0.0010	75.41	74.00	0.0742	0.0126	100.90	100.51	0.7831	0.0019	102.07	102.73	0.5089	0.0047
	0.0005	73.34	73.91	0.5088	0.0047	100.01	100.23	0.7637	0.0021	101.52	102.22	0.8122	0.0017
	0.0002	72.60	72.97	0.5354	0.0044	99.84	100.33	0.7191	0.0025	99.73	98.05	0.2574	0.0080
	0.0001	71.89	71.78	0.7705	0.0021	100.57	99.92	0.7215	0.0025	98.66	98.71	0.9229	0.0007
$d = 4$	0.0100	106.44	104.71	0.2866	0.0075	99.84	99.20	0.3953	0.0060	128.47	126.87	0.6644	0.0031
	0.0050	98.66	97.28	0.1957	0.0091	99.14	98.86	0.6061	0.0036	122.59	120.67	0.6677	0.0030
	0.0020	92.53	91.56	0.2322	0.0084	100.11	100.29	0.8266	0.0015	117.99	116.55	0.4010	0.0059
	0.0010	90.73	88.73	0.2195	0.0087	100.12	100.36	0.5928	0.0038	114.24	113.47	0.7692	0.0021
	0.0005	85.23	86.40	0.3862	0.0061	99.56	99.38	0.1928	0.0092	109.66	109.54	0.5423	0.0043
	0.0002	85.39	83.55	0.0761	0.0125	100.63	100.22	0.5033	0.0047	107.05	107.35	0.8673	0.0012
	0.0001	83.35	81.84	0.1137	0.0112	100.83	101.83	0.5635	0.0041	105.25	104.65	0.8100	0.0017

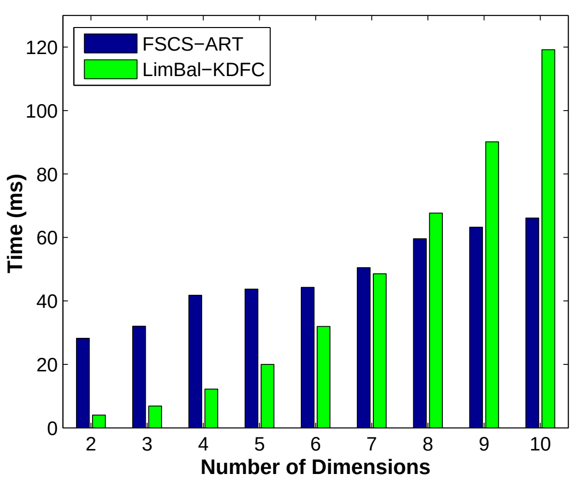
are also consistent. LimBal-KDFC and FSCS-ART have an obvious rise in the F-ratio along with the increase of the number of dimensions.

The failure-detection effectiveness of LimBal-KDFC for $d = 5$ and $d = 10$ is further explored. The corresponding experimental results are shown in Table XI. For block failure patterns, the failure-detection effectiveness of our LimBal-KDFC algorithm is always better than that of the original FSCS-ART for both values of d. The p-values of the rank-sum tests in F-ratios between these two algorithms are always less than 0.05 when $\theta \leq 0.02$. The effect sizes of the differences between the two algorithms are very small in most cases, but it exceeds 0.1 when the failure rate drops to 0.0005 or below. Meanwhile, the difference increases as the number of dimensions increases. For strip patterns, the F-ratios of these two algorithms do not have significant difference for both values of d. For point patterns, the F-ratio of LimBal-KDFC is also always less than that of FSCS-ART. Similar to the results in block failure patterns, the p-values are always less than 0.05 and the effect sizes are still very small in most cases. However, the experimental results show that the effect size of the difference gradually increases as the failure rate decreases or as the number of dimensions increases. For example, when the failure rate drops to 0.0002 in the case of $d = 10$, the effect size is over 0.1. Based on these observations, it is clear that LimBal-KDFC can alleviate the degradation problem of FSCS-ART in detecting failures in block or point patterns for programs with high-dimensional input spaces.

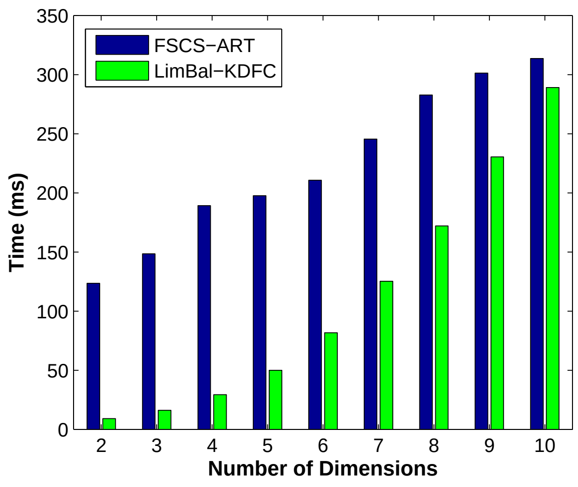
In the original FSCS-ART, test cases are selected strictly according to the max-min distance criterion in equation (1). This test case selection criterion usually incurs a boundary effect [36], and the effect becomes more notable as the number of dimensions increases [37]. However, the approximate nearest neighbor query in our LimBal-KDFC algorithm can



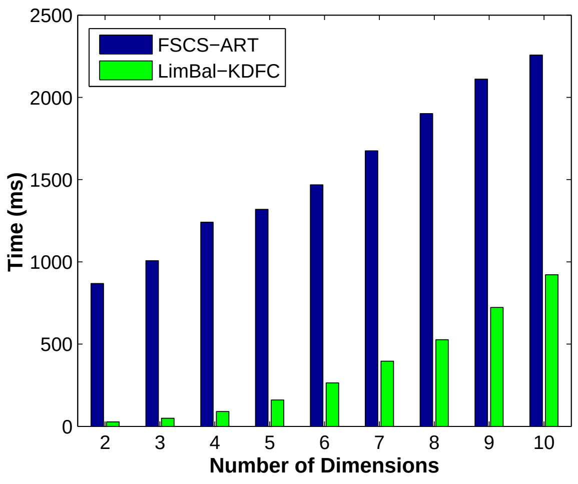
(a) $n = 500$



(b) $n = 1000$



(c) $n = 2000$



(d) $n = 5000$

Fig. 9. Execution times for generating test cases for various dimensions of the input space

reduce the "boundary effect", so the advantage of LimBal-KDFC over FSCS-ART in failure-detection effectiveness is exhibited in the case of high-dimensional input spaces.

Answer to RQ5: Although LimBal-KDFC only realizes the nearest neighbor query approximately, its failure-detection capability has no significant deterioration compared with the original FSCS-ART for low-dimensional input spaces (with $d \leq 4$). More importantly, it exhibits a better performance in failure-detection effectiveness than FSCS-ART for higher-dimensional spaces.

E. Summary and Discussions on Simulation Study

1) Summary on Simulation Study: Based on the above simulation studies, we can confirm that the three KDFC-ART algorithms have realized improvements in efficiency over the original FSCS-ART algorithm in the following two situations: (1) low-dimensional input spaces and (2) large test set sizes (with a small failure rate in the program under test). Specifically, Naive-KDFC stores test cases incrementally to a KD-tree by splitting the input space dimension by dimension in a round-robin fashion. It requires less computation time to generate test cases than the original FSCS-ART for the case of low-dimensional input spaces (with $d \leq 5$) or when the number of test cases is large for high-dimensional spaces, such as $n \geq 5000$ for $d = 10$. SemiBal-KDFC constructs semi-balanced trees by dynamically balancing the dimension splitting. The experimental results show that it can further improve the efficiency of Naive-KDFC for low-dimensional spaces. However, the computation cost for balancing will notably increase as the dimension increases. For $d > 8$, the efficiency of SemiBal-KDFC becomes worse than that of Naive-KDFC. We further find from the experimental results that the Naive-KDFC and SemiBal-KDFC algorithms have better efficiency only when the number of test cases $n \gg 2^d$. When the dimension reaches a limited extent, these two algorithms are useful only when the number of test cases is large. It is, therefore, necessary to further accelerate nearest neighbor queries by applying some limits in backtracking.

LimBal-KDFC is the most efficient among our three KDFC-ART algorithms. It improves the efficiency by imposing a bound on the steps of backtracking for searching the nearest neighbor. For $d \leq 4$, LimBal-KDFC is as efficient as SemiBal-

TABLE XI
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON *F-ratios* BETWEEN LIMBAL-KDFC AND FSCS-ART IN HIGH-DIMENSIONAL INPUT SPACES

Dim.	Failure Rate θ	Block Patterns				Strip Patterns				Point Patterns			
		F-ratio (%)		p-value	Effect size	*F-ratio* (%)		p-value	Effect size	*F-ratio* (%)		p-value	Effect size
		FSCS-ART	LimBal-KDFC			FSCS-ART	LimBal-KDFC			FSCS-ART	LimBal-KDFC		
$d = 5$	0.0100	132.59	127.11	0.0506	0.0138	99.70	99.01	0.6561	0.0031	153.30	148.99	0.0742	0.0126
	0.0050	123.95	119.26	0.0563	0.0135	99.82	98.79	0.0704	0.0128	145.66	141.12	0.1818	0.0094
	0.0020	114.74	109.16	0.0003	0.0258	99.74	101.02	0.7584	0.0022	134.35	129.19	0.0345	0.0149
	0.0010	109.28	104.06	0.0002	0.0261	98.81	98.39	0.8309	0.0015	129.18	123.23	0.0008	0.0238
	0.0005	105.37	99.74	0.0006	0.0242	99.88	101.43	0.5756	0.0040	124.91	118.21	0.0004	0.0253
	0.0002	99.89	95.10	0.0006	0.0244	101.40	101.43	0.8391	0.0014	120.70	115.46	0.0021	0.0217
	0.0001	98.78	94.17	0.0001	0.0276	100.06	100.54	0.6835	0.0037	117.33	112.15	0.0059	0.0195
$d = 10$	0.0100	392.36	380.96	0.0219	0.0162	100.06	100.84	0.4434	0.0054	256.87	241.85	0.0128	0.0244
	0.0050	360.77	339.64	0.0580	0.0134	100.90	100.51	0.5906	0.0038	284.11	265.91	0.0000	0.0470
	0.0020	314.08	271.30	0.0000	0.0591	101.99	101.88	0.7806	0.0020	289.26	265.38	0.0003	0.0358
	0.0010	284.39	232.34	0.0000	0.0821	99.98	100.64	0.5317	0.0044	281.67	248.12	0.0000	0.0624
	0.0005	261.65	205.26	0.0000	0.1446	99.05	99.87	0.6059	0.0036	266.75	231.38	0.0000	0.0751
	0.0002	244.61	182.01	0.0000	0.1975	101.73	100.69	0.4165	0.0057	242.80	200.97	0.0000	0.1046
	0.0001	222.18	166.75	0.0000	0.1849	101.70	100.87	0.6645	0.0031	236.35	191.36	0.0000	0.1391

KDFC. For high dimensions, LimBal-KDFC is more efficient than Naive-KDFC and SemiBal-KDFC. LimBal-KDFC has noticeably alleviated the curse of dimensionality. For example, for the case of $d = 10$, the computation time of LimBal-KDFC is less than that of FSCS-ART when $n \geq 2000$. In addition, since the balancing strategy no longer brings good effect for $d > 8$, LimBal-KDFC can be further speeded up by omitting the balancing strategy in its implementation in the case of high dimensions.

With regard to the failure-detection effectiveness, both Naive-KDFC and SemiBal-KDFC search the exact nearest neighbor for each candidate as FSCS-ART algorithm, so these three algorithms have the same failure-detection capability in theory. Although LimBal-KDFC only finds an approximate nearest neighbor, its failure-detection effectiveness has no significant deterioration compared with the original FSCS-ART. For the case of high dimensions, in fact, LimBal-KDFC exhibits a stronger ability in detecting failures in block and point patterns.

2) Applications of the Algorithms: The adaptive random testing (ART) approach represents a family of algorithms aiming to enhance the failure-detection effectiveness of random testing. Many such algorithms have been developed based on different intuitions to evenly spread the random test cases. These ART algorithms have their distinct effectiveness performance, efficiency performance and characteristics [1]. As a consequence, each ART algorithm has its own favorable and unfavorable scenarios for its application.

The paper by Arcuri and Briand [14] casts doubt on the cost-effectiveness of ART. It points out that the FSCS-ART algorithm requires a considerable amount of time to generate test cases even for very simple programs. In fact, the subject programs in their study have very small failure rates, require large numbers of test cases to detect failures, and the program execution time for each test case is short. It is not appropriate to apply FSCS-ART in such scenarios because of the quadratic complexity in test case generation [13]. On the other hand, their study underscores the need to reduce the test case generation time by FSCS-ART in practical applications, especially when the failure rate is small and the program execution time is short. In the present paper, we aim at taking a step in this important research direction.

Our study reduces the computational overhead by applying KD-trees to support rapid nearest neighbor queries. As shown in the experimental results, the three KDFC-ART algorithms can improve the efficiency in application scenarios involving (1) low-dimensional input spaces and (2) high-dimensional input spaces with low failure rates. Furthermore, in the case of high-dimensional input spaces, LimBal-KDFC can achieve better effectiveness in failure detection than the original FSCS-ART algorithm.

3) Potential Extensions of the Algorithms: Although the proposed KDFC-ART algorithms demonstrate improvements in efficiency, there is further room for improvement in some cases. For example, the advantages of Naive-KDFC and SemiBal-KDFC algorithms are not significant for the case of high-dimensional input spaces. In particular, when testing a program with a high failure rate that requires a small number of test cases, the efficiencies of Naive-KDFC and SemiBal-KDFC may be worse than that of the original FSCS-ART algorithm. In LimBal-KDFC, we adopt a limited backtracking strategy when processing the nearest neighbor query. Although this treatment shows an observable improvement in efficiency for high-dimensional spaces, there are other strategies in KD-trees that may also be adopted to alleviate the curse of dimensionality. For example, best-bin-first search [38] is a heuristic strategy to speed up nearest neighbor queries in KD-trees. A priority queue is used to expedite backtracking. Lengthy searches can also be avoided by fixing the maximum number of leaf nodes explored. In addition, other efficient variants of KD-trees have been proposed, such as multiple randomized KD-trees [39], propagation-assisted KD-trees [40], and randomized KD-forests [41]. These state-of-the-art search algorithms for KD-trees can alternatively be applied to generate test cases efficiently for high-dimensional input spaces when the number of test cases is no more than the order of 2^d.

V. Empirical Studies

A. Experimental Setup

The above simulations have demonstrated that the enhanced FSCS-ART algorithms based on KD-tree have better efficiency than the original FSCS-ART in most cases. Is this conclusion applicable to programs in practice? Thus, it needs to investigate the effect of the three enhanced algorithms in practical applications.

In this section, we report our empirical results on 23 programs that are implemented in Java. The first 12 programs have widely been used in ART studies [13], [42]. They are based on the algorithms published in *Numerical Recipes* [44] and ACM's *Collected Algorithms* [43]. The programs *calDay*, *complex* and *line* are taken from Ferrer et al. [45]. The programs *pntLinePos*, *pntTrianglePos*, and *twoLinesPos* determine the position relationships between a point and a line segment, a points and a triangle, and two line segments, respectively. The *triangle* program classifies a triangle into one of three categories, namely, acute, right, and obtuse. The input parameters of the above four geometric programs are the coordinates of points or the endpoints of line segments. These four programs have been implemented according to the exercises in *Introduction to Java Programming and Data Structures* [46]. The program *nearestDistance* finds the nearest point pair from the five points on the Euclidean plane. The program *calGCD* calculates the greatest common divisor of ten integers. The *select* program [47] returns the k-th largest element from an unordered array. The final program, *tcas*, is an aircraft collision avoidance system created at Siemens [48], which is also used in the experiments of kernel density adaptive random testing (KD-ART) [49] by Patrick et al. The descriptive statistics of these programs are listed in Table XII. Specifically, the dimensions of their input spaces vary from 1 to 12.

We seeded the same mutant faults into the programs for comparing the failure-detection capabilities of the ART algorithms. These faults were generated using the following six common mutation operators [50]: arithmetic operator replacement (AOR), relational operator replacement (ROR), scalar variable replacement (SVR), constant replacement (CR), statement deletion (SDL), and return statement replacement (RSR). The fault types and the total number of faults for each program are shown in the columns of "fault type" and "total faults" in Table XII, respectively. The failure rate of each faulty program was computed in the following way: A huge number of test inputs were continuously picked up in small steps in an almost exhaustive manner from the input domain. They were used to run the program to see whether the execution passes or fails. The failure rate θ could then be calculated as the ratio of the number of failure-causing inputs to the number of all test inputs. The failure rates of all the 23 programs range from 0.0001 to 0.002. Since the input domains of the programs *calGCD*, *select*, and *tcas* are huge, it is impossible to determine their failure rates through exhaustive execution. Thus, their failure rates are marked as "not available" (NA) in Table XII. In the empirical studies, the hardware environment was the same as the above simulation analysis, and the number of repeated trials was set to 5000.

B. Efficiency Comparisons

For the subject programs in this study, we want to investigate the following efficiency problem of three enhanced FSCS-ART algorithms, namely, Naive-KDFC, SemiBal-KDFC and LimBal-KDFC.

RQ6. Are these three enhanced versions of FSCS-ART still efficient to generate test cases for the subject programs?

In the experiments, the time of testing was divided into two parts: the time for generating test cases and the time for executing programs. In general, for the programs under test, the test cases are incrementally generated until revealing the first failure of program. As mentioned earlier, the number of generated test cases is denoted by the F-measure. Thus, the time for generating test cases usually refers to the time of generating all F-measure test cases in the algorithm. Similarly, the test case execution time refers to the total program execution time for all the generated test cases.

As shown in Table XIII, the test case execution time is extremely small and is far less than the test case generation time. Moreover, for each of the subject programs, the test execution times of the four ART algorithms are comparable. From that, it can be seen that the test case generation is the major factor for the efficiency of the whole testing activity of the above 23 programs. That is to say, if the program execution time is very short, it is very important to improve the test case generation efficiency. Accordingly, the exploration in reducing the test case generation time of FSCS-ART is particularly necessary.

When considering the efficiency of test case generation, all the three KDFC-ART algorithms show significant reduction in computation time compared with the original FSCS-ART for all the programs except *nearestDistance* and *calGCD*. The most notable case is the program *probks*, in which the original FSCS-ART algorithm has to use 1388.40 ms to generate test cases before detecting the first failure. However, our Naive-KDFC algorithm takes only 19.48 ms for the same task. The input spaces for the programs *nearestDistance* and *calGCD* are 10-dimensional. Naive-KDFC and LimBal-KDFC exhibit better efficiencies than FSCS-ART for *nearestDistance* while the execution time using SemiBal-KDFC is slightly larger than that of FSCS-ART. Similarly, LimBal-KDFC is more efficient than FSCS-ART in generating test cases for the program *calGCD* while Naive-KDFC and SemiBal-KDFC are slower than FSCS-ART in detecting the first failure. It is not difficult to see that the above observations are in good agreement with previous simulation analysis. In other words, when the number of dimensions is less than or equal to 8, the efficiencies of the three KDFC-ART algorithms is better than that of FSCS-ART; in relatively higher-dimensional input domains, the performance of Naive-KDFC and SemiBal-KDFC is lower than that of FSCS-ART, while LimBal-KDFC can always maintain its advantages in efficiency. Even so, in practical applications, Naive-KDFC and SemiBal-KDFC may still be faster than FSCS-ART, such as for programs *select* and *tcas*.

TABLE XII
DESCRIPTIVE STATISTICS OF THE 23 SUBJECT PROGRAMS

Program	Dim.	Input Space		Size (LOC)	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	0.000716
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
nearestDistance	10	(1,1,1,1,1, 1,1,1,1,1)	(15,15,15,15,15, 15,15,15,15,15)	26	CR	1	0.000256
calGCD	10	(1,1,1,1,1, 1,1,1,1,1)	(1000,1000,1000, 1000,1000,1000, 1000,1000,1000,1000)	24	AOR	1	NA
select	11	(1,1,1,1,1, 1,1,1,1,1,1)	(10,100,100,100,100, 100,100,100,100,100,100)	117	RSR,CR	2	NA
tcas	12	(0,0,0,0,0,0, 0,0,0,0,0,0)	(1000,1,1,50000, 1000,50000,3, 1000,1000,2,2,1)	182	CR	1	NA

Furthermore, the differences between Naive-KDFC, SemiBal-KDFC, and LimBal-KDFC in efficiency are also analyzed in the experiments. In this part of the analysis, we divide the 23 subject programs into two categories. (1) The first category includes programs whose input domains resemble hypercubes, that is, the ranges of all dimensions are equal or comparable. In particular, programs with 1-dimensional input domains are under the first category. Since the semi-balanced strategy is not useful in a 1-dimensional space, and since SemiBal-KDFC and LimBal-KDFC need additional computation costs, their efficiencies are only similar to that of Naive-KDFC, or even a little lower than Naive-KDFC for the case of program *probks*. For programs with 2- to 6-dimensional input domains, SemiBal-KDFC and LimBal-KDFC have an obvious advantage over Naive-KDFC. Meanwhile, the advantage of LimBal-KDFC over SemiBal-KDFC is not observable. LimBal-KDFC may even be a little less efficient than SemiBal-KDFC for some programs with 2- or 3-dimensional input domains such as programs *gammq* and *golden*. For programs with 8- or higher-dimensional input domains, the advantage of SemiBal-KDFC over Naive-KDFC is no longer observable. By contrast, LimBal-KDFC shows very obvious advantages over Naive-KDFC and SemiBal-KDFC. (2) The other category of programs, namely, *bessj*, *gammq*, *plgndr*, *cel*, *calDay*, and *tcas*, do not have input domains that resemble hypercubes. In other words, the ranges of various dimensions of the input domains are quite different. For this category of programs, the efficiencies of SemiBal-KDFC and LimBal-KDFC have greatly improved over that of Naive-KDFC, and the test case generation time has been reduced by 46.1% to 96.7%, On the other hand, the efficiencies of SemiBal-KDFC and LimBal-KDFC are very close to each other.

For programs with input domains that do not resemble hypercubes, an underlying reason for the great improvements of SemiBal-KDFC and LimBal-KDFC over Naive-KDFC lies in the semi-balanced strategy. Unlike round-robin, the semi-balanced strategy prioritizes the long dimensions of the input

TABLE XIII
THE TEST CASE GENERATION TIME AND EXECUTION TIME FOR DETECTING FAILURES IN THE 23 SUBJECT PROGRAMS

Program	Test Case Generation Time (ms)				Test Case Execution Time (ms)			
	FSCS-ART	Naive-KDFC	SemiBal-KDFC	LimBal-KDFC	FSCS-ART	Naive-KDFC	SemiBal-KDFC	LimBal-KDFC
airy	21.07	1.85	1.82	1.84	0.19	0.16	0.17	0.16
bessj0	6.77	0.91	0.89	0.91	0.10	0.09	0.09	0.09
erfcc	330.86	8.54	8.45	8.36	0.84	0.82	0.85	0.85
probks	1388.40	19.48	20.14	20.30	147.65	151.16	149.17	150.18
tanh	3.16	0.58	0.58	0.59	0.03	0.02	0.03	0.02
bessj	10.12	3.74	1.46	1.39	0.46	0.46	0.45	0.43
gammq	67.33	7.81	4.16	4.21	0.37	0.36	0.37	0.38
sncndn	25.85	2.98	2.57	2.51	0.38	0.39	0.39	0.38
golden	200.77	18.28	13.76	13.93	9.14	9.10	8.87	9.04
plgndr	189.12	40.04	7.74	7.76	0.13	0.13	0.13	0.13
cel	233.54	68.34	11.50	11.17	0.15	0.15	0.15	0.15
el2	46.54	11.75	9.15	8.89	0.13	0.18	0.18	0.18
calDay	228.65	76.79	20.27	19.64	2.14	2.23	2.05	2.00
complex	175.58	64.21	57.38	51.38	0.24	0.24	0.25	0.24
pntLinePos	248.83	80.25	70.61	53.79	0.06	0.09	0.08	0.08
triangle	217.72	76.64	66.72	55.86	0.13	0.15	0.16	0.15
line	1786.56	619.62	614.76	366.45	0.23	0.25	0.23	0.23
pntTrianglePos	4038.71	1064.63	1051.80	478.43	1.17	0.90	0.87	0.75
twoLinesPos	12490.38	2439.88	2183.94	890.44	1.37	0.96	0.92	0.78
nearestDistance	743.36	531.58	760.02	324.16	0.35	0.40	0.40	0.38
calGCD	184.16	197.66	270.55	158.27	1.04	1.07	1.11	1.05
select	2215.23	1738.95	1962.32	762.24	1.93	2.10	2.13	1.97
tcas	1486.34	913.29	30.66	29.93	0.21	0.23	0.21	0.20

domains during the splitting process. For KD-trees constructed by this strategy, much fewer nodes will be traversed when finding the nearest neighbors, thus improving the efficiency substantially.

Answer to RQ6: We can conclude from the results of the empirical study that the three KDFC-ART algorithms can significantly reduce the test case generation time for most of the subject programs under study. We also have the following two findings on the comparative efficiencies of the three KDFC-ART algorithms: (1) as the number of dimensions increases, LimBal-KDFC becomes the more obvious winner; (2) for programs with input domains that do not resemble hypercubes, SemiBal-KDFC and LimBal-KDFC show very obvious advantage over Naive-KDFC in efficiency.

C. *Comparisons of Failure-Detection Effectiveness*

We have seen that LimBal-KDFC is more effective than Naive-KDFC and SemiBal-KDFC in reducing the test case generation time for the subject programs, especially for high-dimensional input spaces. However, unlike Naive-KDFC and SemiBal-KDFC, LimBal-KDFC adopts a limited backtracking strategy. Thus, the nearest neighbor query for a given candidate may not necessarily return the actual nearest neighbor. As a consequence, it is important to investigate the failure-detection capability of LimBal-KDFC, and to confirm the following research question:

RQ7. Does LimBal-KDFC have a failure-detection capability on the subject programs comparable to that of FSCS-ART?

During the experimentation, we collected the F-measures of the original FSCS-ART and the LimBal-KDFC algorithms in 5000 trials of the 23 subject programs. The *Wilcoxon rank-sum test* and the corresponding *effect size* analysis were conducted to measure the magnitude of the difference between the two algorithms. From the results in Table XIV, we find that the differences in F-measure between FSCS-ART and LimBal-KDFC measure are quite small for the programs with low-dimensional input spaces (with $\leq$ 4 dimensions). For the programs with relatively higher-dimensional input spaces (with $d \geq 5$), LimBal-KDFC shows an observable advantage over FSCS-ART in some cases.

The F-measures of the original FSCS-ART algorithm are higher than those of LimBal-KDFC except for four programs *tanh*, *bessj*, *sncndn*, and *el2*. For the first 14 programs, namely, from *airy* to *complex*, the differences in F-measures are less than 1.2%. The statistical results about *p*-values and effect sizes show that there is no significant difference between the two algorithms. For the remaining 9 programs, which involve relatively higher-dimensional input spaces, the improvement of LimBal-KDFC becomes observable. Especially for the programs *pntTrianglePos* and *twoLinesPos*, the improvements in the F-measures reach about 10%, and the advantage of LimBal-KDFC over FSCS-ART is with high confidence (see the *p*-values in these two cases). For the remaining seven programs other than the two above, the improvement of LimBal-KDFC relative to FSCS-ART is between 0.7% and 3.1%, and the effect sizes in these cases reflect very little improvement.

TABLE XIV
WILCOXON RANK-SUM TEST AND EFFECT SIZE ANALYSIS ON THE F-MEASURES OF FSCS-ART AND LIMBAL-KDFC

Program	F-measure			p-value	Effect size
	FSCS-ART (x)	LimBal-KDFC (y)	Diff. ($x-y$)		
airy	805.69	802.99	2.70	0.7927	0.0026
bessj0	443.22	442.50	0.72	0.8302	0.0021
erfcc	2890.09	2876.76	13.33	0.7251	0.0035
probks	5571.31	5563.91	7.40	0.9012	0.0012
tanh	308.34	309.43	-1.09	0.9799	0.0003
bessj	446.52	446.79	-0.27	0.8254	0.0022
gammq	1074.84	1073.34	1.50	0.7407	0.0033
sncndn	628.09	633.00	-4.91	0.5993	0.0053
golden	1575.98	1567.08	8.90	0.4413	0.0077
plgndr	1622.64	1609.22	13.42	0.8337	0.0021
cel	1580.78	1567.62	13.16	0.8679	0.0017
el2	708.48	716.13	-7.65	0.7597	0.0031
calDay	1280.65	1265.96	14.69	0.4513	0.0075
complex	1141.65	1137.06	4.59	0.7645	0.0030
pntLinePos	1458.51	1416.60	41.91	0.7828	0.0028
triangle	1419.07	1374.61	44.46	0.1510	0.0144
line	3331.59	3274.84	56.75	0.2593	0.0113
pntTrianglePos	4659.32	4252.41	406.91	0.0001	0.0405
twoLinesPos	7930.68	7082.43	848.25	0.0000	0.0419
nearestDistance	1939.81	1925.87	13.94	0.3293	0.0098
calGCD	1056.18	1026.36	29.82	0.4624	0.0073
select	3292.39	3260.93	31.46	0.2054	0.0127
tcas	2481.62	2419.99	61.63	0.3209	0.0099

Answer to RQ7: Based on the results of the empirical study, we can confirm that LimBal-KDFC has no significant deterioration in failure-detection capability on the subject programs compared with the original FSCS-ART. For some programs with relatively higher-dimensional input spaces (with $d \geq 5$), LimBal-KDFC shows much stronger failure-detection capability than FSCS-ART.

VI. THREATS TO VALIDITY

A. Construct Validity

Construct validity defines how well an experiment measures up to its claims. It refers to whether the operational definition of a variable actually reflects the theoretical concept. F-measure, P-measure, and E-measure, are three common effectiveness metrics used in software testing literature [35]. In particular, F-measure refers to the expected number of tests required to detect the first failure. The research topic in this paper is the incremental generation of test cases by ART, so F-measure is the preferred measure for comparison analysis in our experiments. If our studies were conducted using the other two measures, the comparative results might be different. On the other hand, only a few mutant faults were seeded into the programs under test in the current empirical study. It could also be a potential threat to the validity of the test effectiveness analysis.

Furthermore, the effectiveness of a test algorithm is usually affected by many factors in practical situations, such as the test oracle and failure observations. The current F-/P-/E-measures are simplified but commonly used means to depict the effectiveness. For industrial-scale applications, the effectiveness conclusions based on F-/P-/E-measures need to be further validated by considering the practical factors.

B. External Validity

External validity refers to how well the results of a study can be generalized to other situations. In this study, we have observed that the proposed three KDFC-ART algorithms are suitable for the case of low dimensions and the case of high dimensions with a large test set. Only input spaces with dimensions ≤ 10 are used in our simulation study because the execution times for higher dimensions require more advanced computer configurations. Further studies in higher dimensions may strengthen the comparative results. 23 programs are used as subject programs in the empirical study. Their failure rates cover a wide range including very small values, but their sizes are not large. Further use of other programs with larger sizes may also strengthen the generalization of the experimental results.

On the other hand, we have stated several assumptions in this study, such as programs with numerical inputs, three regular types of failure-causing input regions, and automated detection of test execution failures. To make our ART algorithms applicable to scenarios without the above assumptions, the current algorithms need to be further extended and adapted in the ongoing research.

C. Internal Validity

Internal validity refers to how well the experiments are done. In our study, every experiment has been repeated at least 1000 times to confirm the observations. Different parameter settings may cause a potential threat. For instance, we have set the upper bound for the number of backtrackings according to equation (4). The use of another upper bound may produce different results.

VII. Related Work

As an enhancement of RT, ART can achieve high failure-detection effectiveness through improving the degree of even spreading of the random test cases over the input space [10], [51]. Due to the variety of even-spreading principles, various ART algorithms have been proposed [11]. At the same time, the ART-based tools have been applied to the testing activities for practical applications [52].

Among them, the Fixed-Size-Candidate-Set ART (FSCS-ART) and Restricted Random Testing (RRT) [53] exhibit a relatively higher capability for detecting failures. Although they have better performance on failure detection, they both incur additional computational overheads due to the effort to spread test cases evenly [14], [52]. It is important to reduce the computational overhead without significantly degrading the performance in failure detection.

In fact, some lightweight ART algorithms have been developed. For example, in [54], two methods named Bisection and Random Partitioning have been presented, and their time complexities are $O(n)$ and $O(n \log n)$, respectively [55]. However, their failure-detection capabilities are worse than those of FSCS-ART and RRT. On the other hand, "forgetting" strategy [56] is proposed by Chan et al. to improve the efficiency of ART. Although the computation time of ART methods integrated with this strategy can be reduced to the linear order, the corresponding failure-detection effectiveness also has an observable degradation. Recently, a linear-time ART algorithm ARTsum is proposed by using the category-choice distance measure and the max-sum criterion [57]. The algorithm is mainly used for software with non-numeric inputs. In this paper, our work mainly concerns with the programs with numeric inputs.

Mirroring is another effective way to solve the overhead problem [58]. Although it can significantly improve the efficiency of test case generation, its complexity is still the order of $O(n^2)$ if used with FSCS-ART. Following a similar thought, a divide-and-conquer technique is used to design an efficient ART algorithm by Chow et al. [59]. The input space is partitioned into a lot of sub-spaces, and then the original FSCS-ART is adopted to select test cases in each sub-space. Recently, Huang et al. introduced the "divide-and-conquer" strategy (i.e. dynamic partitioning) into mirror ART [60]. Their approach shows comparable failure-detection capability as the original mirror ART method while having much lower computational overhead. It should be noted that the dynamic partitioning in the above methods is not so natural. It may cause a new problem that how to determine the number of the initial sub-spaces. By contrast, in our three methods, the input space is incrementally partitioned according the executed test cases. More importantly, the information of test cases and sub-spaces is stored in the KD-tree in a natural way. In other words, our enhanced implementations of FSCS-ART do not require any additional information including settings of initial parameters and are more feasible in practice.

As mentioned above, partitioning is an important approach to select test cases for ART, such as the bisection and random partitioning in [54]. In order to facilitate the applications of proportional sampling strategy (PSS) [61], grid partitioning has been widely used in ART methods, and more often is combined with other strategies to reduce the computation time or improve the performance on failure detection [62], [63]. Typically, grid partitioning is utilized to exclude the adjacent cells of a given candidate for selecting test cases as far away as possible in the IP-ART algorithm [64]. This belongs to the exclusion-based ART algorithms, and grid partitioning is mainly used to identify the available cells (i.e., the cells have not been occupied by test cases). Shahbazi et al. proposed the use of centroidal Voronoi tessellations (CVT) to form a random border CVT (RBCVT) algorithm for maximizing the test cases coverage of the input space [9]. In the fast version of RBCVT (known as RBCVT-Fast), grid partitioning is also introduced to reduce the overhead to compute the centroid of background points. There are two limitations of the RBCVT-Fast algorithm. First, the number of test cases needs to be specified before testing. If the specified number is too large, test case generation time may be wasted. If it is too low and no failure is found, a new set of test cases may need to be generated. Second, the failure-detection capability of RBCVT-Fast has a potential risk of instability, especially for programs with high-dimensional input spaces. The underlying reason is that the probability of test cases generated within the "random border" area of the input domain depends on a control parameter. As a result, if the parameters are not chosen fittingly beforehand, the algorithm may not be effective [65].

In our previous work, grid partitioning is also used to "forget" those test cases which are out of "sight" of a given candidate, and then helps to form a linear-order ART algorithm named DF-FSCS [65]. In the above algorithms, although grid partitioning is similarly used to reduce the computational overhead, the number of cells increases exponentially with the number of dimensions in the input space. As a result, this will cause a huge storage cost, especially for high-dimensional input spaces. By contrast, the number of sub-spaces produced by the enhanced ART algorithms based on KD-tree is the same as the number of executed test cases. More importantly, the partitioning in this work is conducted according to the coordinates of test cases, therefore the partitioning action is more natural and direct and can be used for the efficient pruning during the nearest neighbor query.

In addition, quasi-random sequences have been recently used to propose the cost-effective random testing algorithms [42], [66]. In practice, quasi-random sequences are widely utilized to produce low-discrepant sample points, and the low computational cost is the primary advantage of quasi-random testing. In our current work, the data structure of KD-tree is adapted to efficiently realize the nearest neighbor query.

Moreover, the basic methods of tree construction and node backtracking are adapted here to take care of the specific nature of the incremental generation of test cases in ART. Of course, there are many other spatial indexing approaches in computational geometry and computer graphics. For instance, quadtree [67] is often used to recursively divide a two-dimensional space into quadrants. Similarly, octree [68] recursively partitions a three-dimensional space into octants. These two tree structures can also support fast neighbor queries. However, their operations and implementations are more complex than KD-trees. Moreover, it is difficult to enhance them from 2- or 3-dimensional spaces to high-dimensional spaces. R-trees [69] are another effective tree structure for range searches and can also accelerate nearest neighbor queries. Compared with KD-trees, they require more memory for storing regional information and involves more complex insertion operations. In our study, we adapt KD-trees as the data structure for ART based on a comprehensive consideration of simplicity and query efficiency.

VIII. Conclusion

Adaptive random testing realizes a better performance than random testing through an even spreading of random test cases over the input space. Accordingly, it incurs additional computational overhead for the task of evenly spreading test cases. In practice, especially for the programs whose execution time is short, the efficiency of test case generation is as important as the testing effectiveness. As a result, it is important to reduce the computation time of ART without at a loss of its failure-detection capability. In the paper, taking FSCS-ART as an algorithm to be improved because of its popularity and high failure detection capabilities for low dimensional spaces, we proposed three enhanced algorithms by adopting and adapting the KD-tree structure. First, our Naive-KDFC algorithm inserted individual test cases incrementally to a KD-tree by splitting the input space with respect to every dimension in a round-robin manner. Second, a balancing strategy was designed to ensure the partitioning (i.e., the subtrees in KD-tree) as balanced as possible (referred to as SemiBal-KDFC). Subsequently, in order to deal with the notable increase in the number of backtrackings in high dimensions, a limited backtracking strategy was proposed to form the third algorithm LimBal-KDFC. In addition, both simulation analysis and empirical study were performed to validate the efficiencies and effectiveness of our three enhanced algorithms.

According to the results of the simulation analysis, we found that the computation time of FSCS-ART can be significantly reduced through the use of KD-trees for low dimensions and for the case of high dimensions with relatively large number of test cases. More importantly, the Naive-KDFC and SemiBal-KDFC algorithms can preserve the same set of executed test cases as FSCS-ART, and hence the same degree of evenly spreading the random test cases over the input space. At the same time, SemiBal-KDFC can improve the balance of KD-tree and therefore can achieve a better efficiency in test case generation when the dimension of the input space is ≤ 8. In addition, the failure-detection effectiveness of LimBal-KDFC has no significant deterioration compared with FSCS-ART for input spaces with dimensions ≤ 4, but has observable improvement for high dimensions. Based on further empirical results, the advantage of the three KDFC-ART algorithms in terms of efficiency is reconfirmed. Although an approximate nearest neighbor query is adopted in LimBal-KDFC, its failure-detection capability, fortunately, is still comparable to the original FSCS-ART, and even observably better than FSCS-ART for some of the subject programs studied in this paper.

Here, KD-tree has been validated as an effective data structure to alleviate the problem of expensive test case generation for a naive implementation of FSCS-ART. There are still some interesting and important issues that should be deeply studied in the ongoing research. High dimensions may cause problems to both FSCS-ART and the backtracking search in KD-trees. Although our limited backtracking strategy has shown encouraging results, it is worth looking into the feasibility of better methods. On the other hand, how to effectively combine ART with other data structures such as R-trees deserves further investigation. In addition, enhancing the candidate strategy to select multiple test cases in each round is also an interesting research topic.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. The authors also thank Huai Liu of Victoria University in Australia for his expert advice on the statistical analysis. This work was supported in part by the National Natural Science Foundation of China (Grant Nos. 61462030 and 61762040), the Natural Science Foundation of the Jiangxi Province (Grant Nos. 20162BCB23036 and 20171ACB21031), and the Science Foundation of the Jiangxi Educational Committee (Grant No. GJJ150465).

References

[1] S. Anand, E.K. Burke, T.Y. Chen, J. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, 2013, pp. 1978–2001.

[2] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000–2014)," *Proceedings of Future of Software Engineering* (*FOSE 14*), 2014, pp. 117–132.

[3] R. Hamlet, "Random testing," *Encyclopedia of Software Engineering*, 2002.

[4] R. Gerlich, R. Gerlich, and T. Boll, "Random testing: From the classical approach to a global view and full test automation," *Proceedings of 2nd International Workshop on Random Testing (RT 07) (in conjunction with 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 07*)), 2007, pp. 30–37.

[5] J.W. Duran and S.C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, 1984, pp. 438–444.

[6] D. Hamlet and R.N. Taylor, "Partition testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, 1990, pp. 1402–1411.

[7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Object distance and its application to adaptive random testing of object-oriented programs," *Proceedings of 1st International Workshop on Random Testing (in conjunction with 2006 ACM SIGSOFT International Symposium on Software Testing an d Analysis* (*ISSTA 06*)), 2006, pp. 55–63.

[8] A. Arcuri, M.Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, 2012, pp. 258–277.

[9] A. Shahbazi, A.F. Tappenden, and J. Miller, "Centroidal Voronoi tessellations: A new approach to random testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, 2013, pp. 163–183.

[10] T.Y. Chen, F.-C. Kuo, R.G. Merkel, and T.H. Tse, "Adaptive random testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, 2010, pp. 60–66.

[11] T.Y. Chen, F.-C. Kuo, D. Towey, and Z.Q. Zhou, "A revisit of three studies related to random testing," *Science China Information Sciences*, vol. 58, no. 5, 2015, pp. 052104:1–052104:9.

[12] T.Y. Chen and R.G. Merkel, "An upper bound on software testing effectiveness," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 3, 2008, pp. 16:1–16:27.

[13] T.Y. Chen, H. Leung, and I.K. Mak, "Adaptive random testing," *Advances in Computer Science: Proceedings of 9th Asian Computing Science Conference* (*ASIAN 04*), 2005, pp. 320–329.

[14] A. Arcuri and L.C. Briand, "Adaptive random testing: An illusion of effectiveness?" *Proceedings of 2011 International Symposium on Software Testing and Analysis* (*ISSTA 11*), 2011, pp. 265–275.

[15] J.L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of ACM*, vol. 18, no. 9, 1975, pp. 509–517.

[16] A. Andoni and P. Indyk, "Nearest neighbors in high-dimensional spaces," *Handbook of Discrete and Computational Geometry*, 2017.

[17] W.J. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, 1999, pp. 661–674.

[18] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu, "Proportional sampling strategy: Guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, 1996, pp. 775–782.

[19] L.J. White and E.I. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering*, vol. 6, no. 3, 1980, pp. 247–257.

[20] G.B. Finelli, "NASA software failure characterization experiments," *Reliability Engineering and System Safety*, vol. 32, no. 1–2, 1991, pp. 155–169.

[21] M.J.P. van der Meulen, P.G. Bishop, and R. Villa, "An exploration of software faults and failure behaviour in a large population of programs," *Proceedings of 15th International Symposium on Software Reliability Engineering* (*ISSRE 04*), 2004, pp. 101–112.

[22] C. Schneckenburger and J. Mayer, "Towards the determination of typical failure patterns," *Proceedings of 4th International Workshop on Software Quality Assurance (SOQUA 07) (in conjunction with 6th Joint Meeting of European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ES-EC/FSE 07*)), 2007, pp. 90–93.

[23] J. Kubica, J. Masiero, A. Moore, R. Jedicke, and A. Connolly, "Variable KD-tree algorithms for spatial pattern search and discovery," *Proceedings of 18th International Conference on Neural Information Processing Systems* (*NIPS 05*), 2005, pp. 691–698.

[24] A. Adams, N. Gelfand, J. Dolson, and M. Levoy, "Gaussian KD-trees for fast high-dimensional filtering," *ACM Transactions on Graphics*, vol. 28, no. 3, 2009, pp. 21:1–21:12.

[25] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (*HWWS 05*), 2005, pp. 15–22.

[26] J.H. Friedman, J.L. Bentley, and R.A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, no. 3, 1977, pp. 209–226.

[27] D.A. White and R.C. Jain, "Similarity indexing: Algorithms and performance," *Storage and Retrieval for Still Image and Video Databases IV*, 1996, pp. 62–74.

[28] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining*, Pearson Education, 2013.

[29] C. Gini, Variabilità e mutabilità, 1912. Reprinted in *Memorie di Metodologia Statistica*, E. Pizetti and T. Salvemini (eds.), Libreria Eredi Virgilio Veschi.

[30] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, 1945, pp. 80–83.

[31] M. Hollander and D.A. Wolfe, *Nonparametric Statistical Methods (2nd Edition)*, Wiley-Interscience, 1999.

[32] G.M. Sullivan and R. Feinn, "Using effect size – or why the p value is not enough," *Journal of Graduate Medical Education*, vol. 4, no. 3, 2012, pp. 279–282.

[33] J. Pallant, *SPSS Survival Manual–A Step by Step Guide to Data Analysis using SPSS for Windows (3rd Edition)*, Open University Press, 2007, pp. 224–225.

[34] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences (2nd Edition)*, Routledge, 1988.

[35] T.Y. Chen, F.-C. Kuo, and R. Merkel, "On the statistical properties of testing effectiveness measures," *Journal of Systems and Software*, vol. 79, no. 5, 2006, pp. 591–601.

[36] T.Y. Chen, D.H. Huang, T.H. Tse, and Z. Yang, "An innovative approach to tackling the boundary effect in adaptive random testing," *Proceedings of the 40th Annual Hawaii International Conference on System Sciences* (*HICSS 07*), 2007, pp. 262–262.

[37] F.-C. Kuo, T.Y. Chen, H. Liu, and W.K. Chan, "Enhancing adaptive random testing for programs with high dimensional input domains or failure-unrelated parameters," *Software Quality Journal*, vol. 16, no. 3, 2008, pp. 303–327.

[38] J.S. Beis and D.G. Lowe, "Shape indexing using approximate nearest neighbour search in high-dimensional spaces," *Proceedings of 1997 Conference on Computer Vision and Pattern Recognition* (*CVPR 97*), 1997, pp. 1000–1006.

[39] C. Silpa-Anan and R. Hartley, "Optimised KD-trees for fast image descriptor matching," *Proceedings of 2008 IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR 08*), 2008, pp. 1–8.

[40] K. He and J. Sun, "Computing nearest neighbor fields via propagation-assisted KD-trees," *Proceedings of 2012 IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR 12*), 2012, pp. 111–118.

[41] M. Muja and D.G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, 2014, pp. 2227–2240.

[42] T.Y. Chen and R.G. Merkel, "Quasi-random testing," *IEEE Transactions on Reliability*, vol. 56, no. 3, 2007, pp. 562–568.

[43] *Collected Algorithms*, ACM, http://calgo.acm.org/.

[44] W.H. Press, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, Cambridge University Press, 2007.

[45] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Software: Practice and Experience*, vol. 42, no. 11, 2012, pp. 1331–1362.

[46] Y.D. Liang, *Introduction to Java Programming and Data Structures, Comprehensive Version* (*11th Edition*), Pearson Education, 2017.

[47] P.S. May, *Test Data Generation: Two Evolutionary Approaches to Mutation Testing*, PhD Dissertation, The University of Kent, 2007.

[48] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, 2005, pp. 405–435.

[49] M. Patrick and Y. Jia, "KD-ART: Should we intensify or diversify tests to kill mutants?" *Information and Software Technology*, vol. 81, 2017, pp. 36–51.

[50] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, 2011, pp. 649–678.

[51] T.Y. Chen, "Fundamentals of test case selection: Diversity, diversity, diversity," *Proceedings of 2nd International Conference on Software Engineering and Data Mining* (*SEDM 2010*), 2010, pp. 723–724.

[52] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," *Proceedings of 30th International Conference on Software Engineering* (*ICSE 08*), 2008, pp. 71–80.

[53] K.P. Chan, T.Y. Chen, and D. Towey, "Restricted random testing: Adaptive random testing by exclusion," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 4, 2006, pp. 553–584.

[54] T.Y. Chen, G. Eddy, R.G. Merkel, and P.K. Wong, "Adaptive random testing through dynamic partitioning," *Proceedings of 4th International Conference on Quality Software* (*QSIC 04*), 2004, pp. 79–86.

[55] J. Mayer and C. Schneckenburger, "An empirical analysis and comparison of random testing techniques," *Proceedings of 2006 ACM/IEEE International Symposium on Empirical Software Engineering* (*ISESE 06*), 2006, pp. 105–114.

[56] K.P. Chan, T.Y. Chen, and D. Towey, "Forgetting test cases," *Proceedings of 30th Annual International Computer Software and Applications Conference* (*COMPSAC 06*), 2006, pp. 485–494.

[57] A.C. Barus, T.Y. Chen, F.-C. Kuo, H. Liu, R.G. Merkel, and G. Rothermel, "A cost-effective random testing method for programs with non-numeric inputs," *IEEE Transactions on Computers*, vol. 65, no. 12, 2016, pp. 3509–3523.

[58] T.Y. Chen, F.-C. Kuo, R.G. Merkel, and S.P.H. Ng, "Mirror adaptive random testing," *Information and Software Technology*, vol. 46, no. 15, 2004, pp. 1001–1010.

[59] C. Chow, T.Y. Chen, and T.H. Tse, "The ART of divide and conquer: An innovative approach to improving the efficiency of adaptive random testing," *The Symposium on Engineering Test Harness* (*TSETH 13*),

Proceedings of 13th International Conference on Quality Software (*QSIC 13*), IEEE Computer Society, 2013, pp. 268–275.

[60] R. Huang, H. Liu, X. Xie, and J. Chen, “Enhancing mirror adaptive random testing through dynamic partitioning,” *Information and Software Technology*, vol. 67, no. C, 2015, pp. 13–29.

[61] T.Y. Chen, T.H. Tse, and Y.T. Yu, “Proportional sampling strategy: A compendium and some insights,” *Journal of Systems and Software*, vol. 58, no. 1, 2001, pp. 65–81.

[62] J. Mayer, “Lattice-based adaptive random testing,” *Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 05*), 2005, pp. 333–336.

[63] A. Gotlieb and M. Petit, “A uniform random test data generator for path testing,” *Journal of Systems and Software*, vol. 83, no. 12, 2010, pp. 2618–2626.

[64] T.Y. Chen, D.H. Huang, and Z.Q. Zhou, “On adaptive random testing through iterative partitioning,” *Journal of Information Science and Engineering*, vol. 27, no. 4, 2011, pp. 1449–1472.

[65] C. Mao, T.Y. Chen, and F.-C. Kuo, “Out of sight, out of mind: A distance-aware forgetting strategy for adaptive random testing,” *Science China Information Sciences*, vol. 60, 2017, pp. 092106:1–092106:21.

[66] H. Liu and T.Y. Chen, “Randomized quasi-random testing,” *IEEE Transactions on Computers*, vol. 65, no. 6, 2016, pp. 1896–1909.

[67] R.A. Finkel and J.L. Bentley, “Quad trees a data structure for retrieval on composite keys,” *Acta Informatica*, vol. 4, no. 1, 1974, pp. 1–9.

[68] D. Meagher, “Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-D objects by computer,” *Technical Report IPL-TR-80-111*, Rensselaer Polytechnic Institute, 1980.

[69] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (*SIGMOD 84*), 1984, pp. 47–57.