KD-RRT: Restricted Random Testing based on K-Dimensional Tree

Junlong Lian[†], Chenhui Cui[†], Weifeng Sun[†], Yiming Wu[†], Rubing Huang[†,‡]
[†]*School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, Jiangsu 212013, China*
[‡]*Faculty of Information Technology, Macau University of Science and Technology, Taipa, Macau 999078, China*
Email: 2211908018@stmail.ujs.edu.cn

***Abstract*—*Adaptive random testing* (ART) improves the failure-detection capability of *Random Testing* (RT) by making the generated test cases as evenly distributed as possible within the input domain. *Restricted Random Testing* (RRT) is one of the most classical ART algorithms, which defines an exclusion region around each executed test case, and generates subsequent test cases from outside all the exclusion regions. RRT requires calculating the distances to the executed test cases sequentially for each candidate test case. When the number of executed test cases is large, the time overhead will be very high. In this paper, we propose a method to reduce the time overhead of RRT by using a *K-Dimensional Tree* (KD-Tree), called KD-RRT. KD-RRT only calculates the distances of executed test cases that are close to the candidate test cases, and ignores the test cases that are farther away. Thus, KD-RRT cuts down the number of distance calculations and reduces the time overhead of RRT. To verify the effectiveness and efficiency of KD-RRT, a series of simulations and empirical studies are designed in this paper. The data results show that KD-RRT ensures the effectiveness of RRT and significantly reduces the time overhead of RRT.**

***Index Terms*—random testing, adaptive random testing, restricted random testing, KD-Tree, nearest neighbor search**

I. Introduction

With the increasing scale and complexity of software, software testing has become increasingly important in software development, and automated testing has received more and more attention. *Random Testing* (RT) [1], [2] is a basic testing technique with the advantages of low cost, fast and ease of implementation. RT is a black-box testing method that randomly generates one test case at a time from the *input domain* (the set of all possible inputs to the program) of a program and executes it until the termination condition is met. RT has been widely used in many practical systems, such as UNIX programs [3], Java just-in-time compilers [4], Windows NT applications [5], embedded software systems [6], SQL database systems [7] and Android applications [8].

Although RT is well suited for automated testing, RT has disadvantages such as blindness, low coverage and leading to many redundant tests. *Adaptive Random Testing* (ART) [9] is one approach to enhance RT. ART is motivated by the observation of software failure behavior and patterns independently reported by many researchers in several different fields [10]. The program inputs that trigger failures (*failure-causing inputs*) tend to cluster into contiguous regions (*failure regions*) [11]–[15]. Many ART methods [16]–[20] have been proposed and widely used.

Restricted Random Testing (RRT) [21] is one of the most classic ART implementations. RRT achieves an even distribution of test cases by defining exclusion regions for each executed test case and finding new test cases to be executed outside the exclusion regions. Previous studies have shown that RRT can effectively improve the failure-detection capability of RT and reduce the number of test cases that find failures [22]. A straightforward implementation of RRT requires calculating the distances among each randomly generated candidate test case and all executed test cases. And it compares the distances with the radius of the exclusion regions. If there is a distance less than the exclusion radius the candidate test case is discarded and regenerated. The process of selecting the next test case to be tested will result in a significant time overhead, so it is necessary to reduce the time overhead of RRT without affecting the failure-detection capability of RRT.

As mentioned above, the primary time overhead of RRT is to compute the distances to the executed test cases in turn for the candidate test cases and then compare with the exclusion radius. This problem can be transformed into a problem of finding the nearest neighbors for the candidate test cases from the set of executed test cases. As long as a distance smaller than the exclusion radius appears in the process of finding the nearest neighbor, the candidate test case is directly discarded. Therefore, to solve this problem, it is essential to search the nearest neighbor for candidate test cases more intelligently.

In this paper, we propose to use *K-Dimensional Tree* (KD-Tree) [23] to improve the efficiency of RRT, namely *restricted random testing based on KD-Tree* (KD-RRT). Our motivation is to speed up nearest neighbor calculations by using KD-Tree, and ultimately achieve the goal of reducing RRT time overhead.

To evaluate the effectiveness and efficiency of KD-RRT, we designed a series of simulations and empirical studies on 20 real-life programs. The main contributions of this paper are summarized as follows:

- We propose an improved RRT algorithm based on KD-Tree, i.e., KD-RRT.
- We report simulations and empirical studies of KD-RRT in terms of test effectiveness (i.e., failure-detection capability) and test efficiency (i.e., test case generation time).
- Compared with RRT, our proposed approach significantly reduces the time overhead while ensuring fairly compa-

rable failure-detection.

The remainder of this paper is organized as follows: Section II briefly describes the background of failure patterns, RRT and provides an overview of KD-Tree. Section III discusses the details and complexity analysis of KD-RRT. Section IV presents the experimental design. Section V shows the experimental results and discusses the experimental results in detail. Section VII concludes the paper.

II. Background

We introduce the failure patterns, the original RRT and KD-Tree in this section.

A. Failure Patterns

For a given *Software/System Under Test* (SUT), the *failure-causing input* refers to the program input that triggers the SUT output or the behavior does not meet the test expectations (*test oracle* [24]). The failure region is the set of all failure-causing inputs. In general, there are two basic characteristics to describe a failure region: *failure pattern* and *failure rate*. The failure pattern describes the shape of the failure regions and their locations in the input domain.Besides, the failure rate refers to the size of the failure region as a percentage of the size of the input domain, which is denoted by θ.

From previous studies [11], [13], [15], it is easily investigated that the failure region is continuous and the non-failure region is also continuous. Chan et al. [25] classified the failure patterns into three main types: block patterns, strip patterns and point patterns. Figure 1 provides an example of these three failure patterns in a 2-dimensional input domain, where the square bounding box represents the boundary of the input domain; the black blocks, bars and dots represent the inputs that cause failures. Previous studies [11], [13], [15] have shown that strip and block patterns are more common than point patterns in real programs.

B. Restricted Random Testing

The original RRT is a restriction-based ART algorithm. There are two versions [21] of the original RRT: Ordinary RRT (ORRT) and Normalized RRT (NRRT). These two versions share the same fundamental algorithm, but differ in processing non-homogeneous input domains. The RRT discussed in this paper is ORRT.

RRT defines an exclusion region around each test case that has been executed but no failure has been found, and

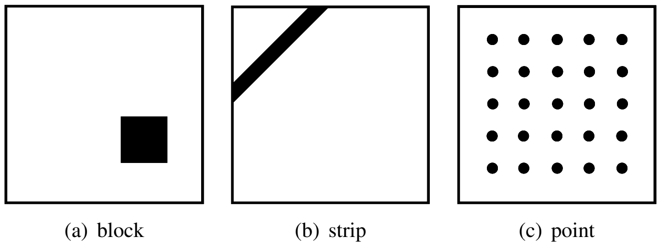


Fig. 1. three failure patterns in a 2-dimensional input domain

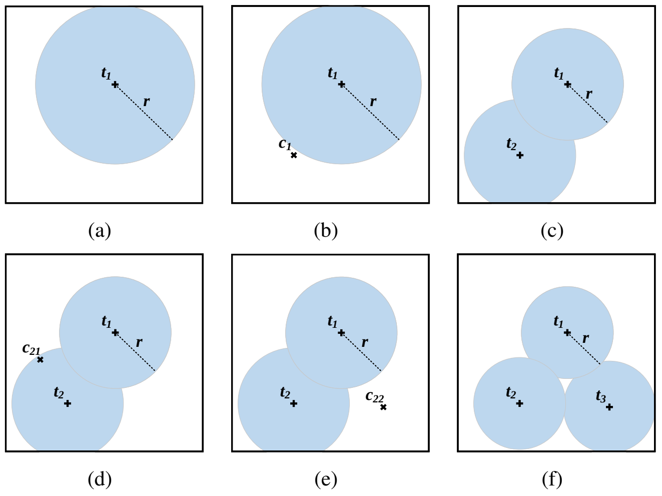


Fig. 2. Example of generating the first few test cases in 2-dimension with RRT.

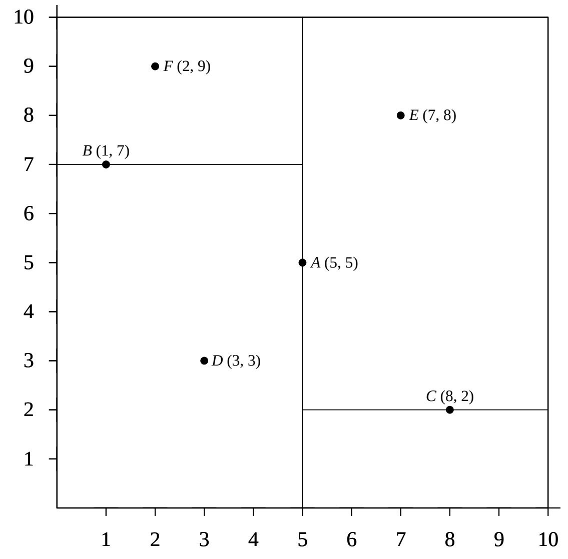
subsequent test cases cannot be generated in the exclusion region. Each exclusion region can be seen as hyperspheres in different dimensional input domains. Let d be the dimension of the input domain D, then the radius r of the exclusion regions is defined as follows:

$$r = \sqrt[d]{\frac{C_d \times S \times R}{|E| \times \pi^{\lfloor d/2 \rfloor}}} \tag{1}$$

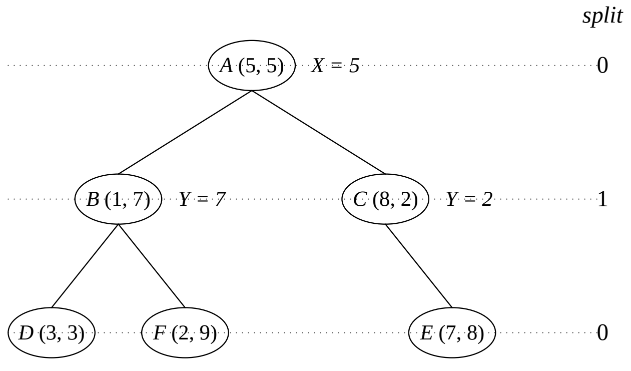
where S is the size of the input domain, R indicates the exclusion rate (set by the tester, in this paper, the exclusion rate $R = 0.75$ is set according to the recommendation of Chan et al. [26]), $|E|$ refers to the number of the executed test cases. And C_d is the formula coefficient, which is defined as follows:

$$C_d = \begin{cases} \frac{C_{d-2} \times d}{2} & , \quad d > 2 \\ 1 & , \quad d = 2 \\ 1/2 & , \quad d = 1 \end{cases} \tag{2}$$

An example of using the RRT algorithm to generate the first few test cases in 2-dimension is shown in Figure 2. The square box in the figure indicates the input domain, "$+$" indicates the test case that was executed but no failure aws found, "$\times$" indicates the candidate test case, and the shading indicates the exclusion region. RRT randomly generates the first test case t_1 from the input domain, and then calculates the radius r of the exclusion region according to equation (1), i.e., $\sqrt{R * S/\pi}$. So as to determine the exclusion region, as shown in Figure 2(a). After that, RRT will randomly generate a candidate test case from the input domain, assuming the generated candidate test case is c_1, as shown in Figure 2(b). The distance between c_1 and t_1 is calculated as $d(c_1, t_1)$. Obviously, $d(c_1, t_1)$ is greater than the exclusion radius r, which shows that c_1 is located outside the exclusion region, so c_1 can be executed as the next test case. After c_1 has been executed, it will be added into $|E|$ as t_2. Then r will be updated as shown in Figure 2(c). Suppose the next generated candidate is c_{21} in Figure 2(d). It is known that c_{21} falls in the exclusion region



(a) Sub-space partitioning process of KD-Tree



(b) Data structure of KD-Tree

Fig. 3. Example for KD-Tree in a 2-dimensional space

by $d(c_{21}, t_2) < r$, so the algorithm needs to discard c_{21} and regenerate the candidate test cases. The new candidate test case is c_{22} in Figure 2(e). Then $d(c_{22}, t_1)$ and $d(c_{22}, t_2)$ are calculated and compared with r. It can be seen that c_{22} falls outside the exclusion region, so c_{22} will be executed as t_3. The updated r is shown in Figure 2(f).

C. KD-Tree

KD-Tree [23], [27]–[31] is a data structure that uses the idea of branch reduction to reduce unnecessary distance calculations when computing nearest neighbor, thus improving the efficiency of nearest neighbor computations. Moreover, the KD-Tree is widely used in many scenarios [32]–[34] due to its high-speed query efficiency and simple in structure [35].

The KD-Tree was first proposed by Bentley [23] to solve the best matching problem, and the KD-tree is a binary tree [36]. The contents in each node in the KD-tree are shown in Table I. The set of nodes in the KD-Tree represents a sample set, and each node represents a sample. In this paper, the sample set is the set of executed test cases, and each sample is an executed test case. In each node, *data* field holds a d-dimensional array, which represents a point in the d-dimensional space, i.e., a test case. *split* field holds integer data, which represents the hyperplane, i.e., the dimension to which the current node is split. And *parent* field holds the parent of the current node, which is convenient for backtracking when querying.

TABLE I
CONTENT OF NODES IN THE KD-TREE

Field Name	Field Type	Description
data	Arrays	A point from d-dimensional space
split	integer	The splitting dimension.
left	kd-tree node	The root node of a kd-tree representing those points to the left of the splitting plane.
right	kd-tree node	The root node of a kd-tree representing those points to the right of the splitting plane.
parent	kd-tree node	Parent node of current node.

Define the d keywords of *data* in node P as $K_0(P), \ldots, K_{d-1}(P)$. Node P divides the space into two sub-spaces, left and right, sub-space according to the *split* value. Suppose *split* of node P is i. Then for any node Q in the sub-tree of node P, Q lies on the left side of P if and only if $K_i(Q) < K_i(P)$, and the point Q lies on the right side of P if and only if $K_i(Q) > K_i(P)$. All nodes at a given level in a KD-Tree have the same *split*. We generally consider the root node to be at level 0. The *split* of the root node is 0, the *splits* of its two children nodes are 1, and so on until the $(d-1)th$ level, whose *splits* are $(d-1)$; the *splits* of the nodes at the dth level are 0, and then the cycle repeats. The *split* of the node at level j is determined by the function $Split(j)$, which is defined as follows:

$$Split\,(j) = j\%d \tag{3}$$

Take a set of points $\{A(5,5), B(1,7), C(8,2), D(3,3), E(7,8), F(2,9)\}$ in a 2-dimensional space as an example. Figure 3(a) demonstrates the sub-space partitioning process of this group of points based on the KD-Tree. Figure 3(b) shows the corresponding data structure. The right side of Figure 3(b) shows the *split* of the nodes on the corresponding level. In sequential order, point A is taken as the root node. The *split* of A is 0, which means that $X = 5$ is the first hyperplane and divides the space into two sub-spaces. For point B, with $K_0(B) = 1 < 5$, point B is in the left sub-tree of A. For point C, $K_0(C) = 8 > 5$, so point C is on the right sub-tree of A. B and C are sub-nodes of A, so their *splits* are 1. Similarly, point D is on the left sub-tree of B, point F is on the right sub-tree of B, point E is on the right sub-tree of C. The *splits* of D, E and F are all 0.

III. KD-RRT

Our method reduces the time overhead of RRT by using KD-Tree to compute the nearest neighbor of candidate test cases. This section provides an introduction to the algorithm, space complexity and time complexity.

A. Algorithm

Algorithm 1 describes the detailed information about KD-RRT. The structure of each node in the KD-Tree is as discussed in Table I. *getRandomTC(D)* in lines 2 and 8 is a function that randomly generates a test case from the input domain D. The *termination condition* in line 4 of the algorithm can be a failure found, a certain coverage rate achieved, or a certain number of executed test cases generated, etc. After accepting the required parameters for the algorithm and initializing the KD-Tree E and exclusion radius r (line 1), the first step is to randomly generate the first test case tc from the input domain (line 2). KD-RRT then inserts tc into E as the root node, and set *split* of the root node to 0 after testing the program with tc (line 3).

Algorithm 1 KD-RRT

```
Input: 1) Target exclusion ratio $R$;
       2) Input domain $D$;
Output: The set of test cases $E$ (or KD-Tree);
 1: Set the KD-Tree $E = \{\}$, set exclusion radius $r = 0$;
 2: Let test case $tc = getRandomTC(D)$, and test the program with $tc$ as the first test case;
 3: Insert $tc$ into $E$ as the root node, set split of root node to 0;
 4: while termination condition does not satisfy do
 5:     Determine a circular exclusion region around each executed test case in $E$, and update $r$ with equation (1);
 6:     Set $outside = false$;
 7:     while not $outside$ do
 8:         $tc = getRandomTC(D)$, $outside = true$;
 9:         Find the smallest space containing $tc$ according to the split of nodes in $E$, let node $p_{tc}$ = the node that divides this smallest space;
10:         Calculate the nearest neighbor of $tc$ by backtracking $E$ from $p_{tc}$, and if one of the distances during the backtracking is less than $r$, then $outside = false$ and break the backtracking;
11:     end while
12:     Test the program with $tc$;
13:     if $K_{p_{tc}.split}(tc) < K_{p_{tc}.split}(p_{tc}.data)$ then
14:         Insert $tc$ as the left child of node $p_{tc}$.
15:     else
16:         Insert $tc$ as the right child of node $p_{tc}$.
17:     end if
18: end while
19: return $E$
```

Lines $4-18$ is the core of KD-RRT. KD-RRT will stop when the termination condition is met.

In each turn, KD-RRT generates a test case and inserts it into the KD-Tree after being executed but without causing any failure in the software. During the loop, KD-RRT first updates r with equation (1) (line 5). Lines $6-12$ is to generate a test case which is outside of all the exclusion regions. KD-RRT no longer calculates the distances among the generated candidate test case and the executed test cases on a rotating basis like the original RRT.

KD-RRT determines whether a randomly generated test case tc should be dropped in two main steps. The first step is finding the node p_{tc} that belongs to the same minimal space as tc (line 9). The second step is starting from the found node p_{tc} and backtracking to find the nearest neighbor or the first distance less than r (line 10).

The first step also prepares for inserting the test case into the KD-Tree. Specifically, KD-RRT starts from the root node and finds downward step by step using the split hyperplane determined by *split* in each node. That is, from the root node a, KD-RRT decides whether to search to the left sub-space or to the right sub-space by comparing $K_{a.split}(a.data)$ and $K_{a.split}(tc)$. And if $K_{a.split}(a.data) > K_{a.split}(tc)$ then the left sub-tree is searched, i.e., set $a = a.left$, otherwise the right sub-tree is searched, i.e., set $a = a.right$. Then iterates through the above comparison until a is a empty pointer and stores the last node that is not null as p_{tc}. If tc is not discarded, p_{tc} will be the parent node of tc.

The backtracking process is terminated and the candidate test case is regenerated once a distance less than r occurs. In detail, KD-RRT firstly calculates the distance between tc and p_{tc}, and defines p_{tc} as temporary nearest neighbor. KD-RRT then performs a recursive upward backtracking and does the following for the current node *cp*. Compare the distance from tc to the split hyperplane of *cp* as $d(tc, cp.split)$ with the distance of the current temporary nearest neighbor as dis_{min}. If $d(tc, p.split) < dis_{min}$, it means that there may be a set of points T in the region on the other side of *cp* that are smaller than dis_{min}, specially, $cp \in T$. Therefore, KD-RRT need to calculate the distance between tc and *cp* as $d(tc, cp)$. If $d(tc, cp) < dis_{min}$, the temporary nearest neighbor will be updated to *cp*, and dis_{min} will be updated to $d(tc, cp)$. Also, the sub-tree formed by the other side of *cp* needs to be searched recursively. If $d(tc, cp.split) > dis_{min}$, it means that $d(tc, cp)$ and the distances among tc and the points on the other side of *cp* are definitely larger than dis_{min}. So there is no need to calculate the $d(tc, cp)$ and the distances among tc and the points on the other side of *cp* but just back up. The search process ends when the backtracking reaches the root node. The last temporary nearest neighbor is the nearest neighbor of tc. During the backtracking process, KD-RRT compares r with dis_{min} at each update of temporary nearest neighbor. The backtracking will be terminated and tc will be regenerated once $r > dis_{min}$.

B. Complexity Analysis

During the procedure of KD-RRT, memory space is required to store the KD-Tree. For a KD-Tree containing n executed test cases, $O(n)$ space is required to store the executed test cases and the space for the tree structure and *split* is proportional to n. So the final space complexity of KD-RRT is $O(n)$.

In our method, a lot of previously executed test cases are filtered out for distance calculations. To get the nth test case, RRT needs to generate an average of $logn$ candidate test cases for each previously executed test case [22]. The time required for RRT to generate the nth test case is $O(nlogn)$. So the time complexity of RRT to generate n test cases is $O(n^2logn)$. For KD-RRT, the average time complexity of computing the nearest neighbor of a candidate test case is $O(logn)$ by a KD-Tree containing n executed test cases. So the time required for KD-RRT to generate the nth test case is $O(log^2n)$. So the time complexity of KD-RRT to generate n test cases is $O(nlog^2n)$.

IV. Evaluation

This paper aims to enhance the efficiency of RRT. Therefore, we focus on the comparison with RRT in this study. We conduct a series of simulations and empirical studies to evaluate the performance of KD-RRT. The design and settings of the experiments are described in this section.

A. Research Questions

We propose KD-RRT to reduce the time overhead of RRT further. Therefore, it is required to examine the test case generation time of KD-RRT. In addition, while decreasing the time overhead, we hope that KD-RRT also has comparable failure-detection effectiveness to RRT to deliver high cost-effectiveness in testing. Thus, it is also essential to evaluate the failure-detection effectiveness of KD-RRT under different scenarios. Thus, our experimental studies help us answer the following two research questions.

RQ1: How effective is KD-RRT to detect the failure?

RQ2: To what extent does KD-RRT reduce the high time overhead of RRT?

B. Variables and Measures

RQ1 evaluates and compares the failure-detection effectiveness of the test techniques. We use the *F-measure* [9], [17], [26], [37], which refers to the expected number of test cases required to detect the first failure, to measure the effectiveness of the algorithms. In our study, F_{RT} denotes the F-measure of RT. According to uniform distribution, theoretically, $F_{RT} = \frac{1}{\theta}$. F_{ART} denotes the F-measure of ART. And another metric *F-ratio* denotes the ratio of F_{ART} to F_{RT}, which measures the F-measure improvement of ART over RT. Therefore the smaller the F-ratio, the stronger the effectiveness of the corresponding ART algorithm.

RQ2 is mainly to compare the execution time of KD-RRT and RRT. Two main aspects are included. In the simulations, the time required to generate a certain number of test cases as *generation time* is used. In the empirical studies, the time required to find the first failure as *F-time* is used.

In this paper, we also employ two statistical measures, *p-value* and *effect size* [38], to measure the performance differences between KD-RRT and RRT. The *Wilcoxon rank-sum test* [39] is used to verify whether there are differences in efficiency among the investigated ART algorithms. If the p-value is less than 0.05, it means that there is a significant difference between the two methods being compared. Meanwhile, the effect size is used to measure the magnitudes of the differences among ART algorithms. For two methods X and Y, effect size $= 0.5$ indicates the two algorithms perform similarly. While effect size greater than 0.5 suggesting that X is superior to Y.

C. Simulations and Object Programs

To address RQ1, we attempt to examine and compare the values of F_{ART} for RRT and KD-RRT. F_{ART} has been influenced by many factors, such as dimension, the number and the compactness of failure regions, as well as the existence and the size of a predominant failure region, so theoretical analysis of F_{ART} is of great difficulty. In previous studies [9], [26], [40], researchers commonly used simulations or empirical studies to investigate F_{ART}. In our study, we conduct both simulations and empirical studies to evaluate the failure-detection effectiveness.

To address RQ2, we conduct simulations to evaluate the test case generation time of KD-RRT compared to the original RRT. In the simulations, we use RRT and KD-RRT to generate a total of 15,000 test cases from 1-dimensional to 10-dimensional input domains. The execution time taken to generate n test cases ($n = 1000, 2000, 3000, \ldots, 15000$) is recorded and thus compared. In addition, in the empirical studies, we compare KD-RRT and RRT based on the average time taken to detect the first failure.

The parameters of the simulations are set as follows.

- dimension d: 1, 2, 3, 4, 5, 6, 8, 10;
- failure rate θ: 0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002, 0.0001;
- failure patterns: black, strip, point;
- number of test cases to generate: 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000, 12000, 13000, 14000, 15000;
- domain boundary: $(-5000, 5000)^d$, which means that the boundary of each dimension of the input domain is set from -5000 to 5000.

In this study, we collect a total of 20 real-life programs as object programs to conduct the empirical studies. The first 12 programs, which are taken from ACM's collected algorithms [41] and Numerical Recipes [42], have been widely used in previous ART studies [9], [16], [17], [26]. The programs *calDay*, *complex*, and *line* are taken from [43]. The programs *pntLinePos*, *pntTrianglePos*, *twoLinesPos* and *triangle* have been implemented according to [44]. The *nearestDistance* program takes five points in 2-dimensional space and returns the two points that are nearest to each other [18]. The first

TABLE II
INFORMATION OF OBJECT PROGRAMS

No.	Program	Dim.	Input Space		Fault Type	Total Faults
			From	*To*		
1	airy	1	(-5000.0)	(5000.0)	CR	1
2	bessj0	1	(-300000.0)	(300000.0)	AOR, ROR, SVR, CR	5
3	erfcc	1	(-30000.0)	(30000.0)	AOR, ROR, SVR, CR	4
4	probks	1	(-50000.0)	(50000.0)	AOR, ROR, SVR, CR	4
5	tanh	1	(-500.0)	(500.0)	AOR, ROR, SVR, CR	4
6	bessj	2	(2, -1000)	(300, 15000)	AOR, ROR, CR	4
7	gammq	2	(0, 0)	(1700, 40)	ROR, CR	4
8	sncndn	2	(-5000, -5000)	(5000, 5000)	SVR, CR	5
9	golden	3	(-100, -100, -100)	(60, 60, 60)	ROR, SVR, CR	5
10	plgndr	3	(10, 0, 0)	(500, 11, 1)	AOR, ROR, CR	5
11	cel	4	(0.001, 0.001, 0.001, 0.001)	(1, 300, 10000, 1000)	AOR, ROR, CR	3
12	el2	4	(0, 0, 0, 0)	(250, 250, 250, 250)	AOR, ROR, SVR, CR	9
13	calDay	5	(1, 1, 1, 1, 1800)	(12, 31, 12, 31, 2200)	SDL	1
14	complex	6	(-20, -20, -20, -20, -20, -20)	(20, 20, 20, 20, 20, 20)	SVR	1
15	triangle	6	(-25, -25, -25, -25, -25, -25)	(25, 25, 25, 25, 25, 25)	CR	1
16	pntLinePos	6	(-25, -25, -25, -25, -25, -25)	(25, 25, 25, 25, 25, 25)	CR	1
17	line	8	(-10, -10, -10, -10, -10, -10, -10, -10)	(10, 10, 10, 10, 10, 10, 10, 10)	ROR	1
18	pntTrianglePos	8	(-10, -10, -10, -10, -10, -10, -10, -10)	(10, 10, 10, 10, 10, 10, 10, 10)	CR	1
19	twoLinesPos	8	(-15, -15, -15, -15, -15, -15, -15, -15)	(15, 15, 15, 15, 15, 15, 15, 15)	CR	1
20	nearestDistance	10	(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	(15, 15, 15, 15, 15, 15, 15, 15, 15, 15)	CR	1

12 programs are implemented in C++, while the remaining 8 are in Java. The programs vary in dimension 1 to 10, and are seeded with faults using the following mutation operators [45] : arithmetic operator replacement (AOR), relational operator replacement (ROR), Scalar variable replacement (SVR), Constant repleacement (CR), statement deletion (SDL). The detailed information of these programs is shown in Table II.

According to the central limit theorem [46], all experiments were repeated a sufficient number of times S. In simulations, S is set to 3000 for calculating F-ratio, and S is set to 1000 for calculating generation time. In empirical studies, S is set to 3000 for calculating F-measure and F-time. All the result data discussed in section V are the average of repeated execution.

D. Experiment environment

All the experiments are run on laptop, Intel(R) Core(TM) i5-10210U CPU (1.60GHz, 4 cores, 8 logical processors) with 16GB of RAM running under the 64-bit Windows 10 operating system. And both RRT and KD-RRT are implemented in Java and executed on java 1.8.

V. RESULTS

A. Answer to RQ1

Table III shows the F-ratio comparisons between KD-RRT and RRT for simulations with block patterns, strip patterns and point patterns. Based on the experimental data, we can know that KD-RRT has a similar failure-detection capability to RRT. The p-values are all greater than 0.05 except for a few less than 0.05 (e.g., $d = 2$, $\theta = 0.01$, strip patterns, and $d = 4$, $\theta = 0.001$, point patterns). This implies that the data from KD-RRT and RRT are not significantly different. In addition, the effect sizes are all around 0.5, which indicates that KD-RRT and RRT are similar.

Figure 4 shows the F-measure results of empirical studies. It can be clearly seen that for each real program, the F-measures of KD-RRT and RRT are similar. The results are comparable to those of the simulations.

Overall, compared to RRT, KD-RRT has comparable failure-detection effectiveness.

B. Answer to RQ2

Figure 5 shows the comparison of the time required to generate a set number of test cases between KD-RRT and RRT for different dimensions. The x-axis represents the number of test cases to generate, while the y-axis represents the time required to generate the corresponding number of test cases. In addition, Table IV shows partially detailed generation time comparisons between KD-RRT and RRT for simulations. It can be seen that KD-RRT has a much lower generation time than RRT below the 10-dimensional input domain, regardless of dimensions. From the data in the table, it is obvious that all p-values are 0 and all effect sizes are much larger than 0.5. This indicates that the time overhead of KD-RRT is much lower than that of RRT. In other words, KD-RRT is much better than RRT in terms of efficiency.

It is quite intuitive from the figure that the time overhead of RRT is not linear regardless of the dimension. The time overhead of KD-RRT is approximately linear in magnitude at $d \leq 6$. But the time overhead of KD-RRT increases as the dimension increases. This is mainly due to the shortcoming of the original KD-Tree. Specifically, KD-Tree requires backtracking when searching for the nearest neighbor. Backtracking entails finding possible nearest neighbors in sub-tree branches that have not been visited and that intersect the hypersphere of target point Q. As the dimension d increases, the number of hyper-rectangles (the area where the sub-tree branches are located) intersecting the hypersphere of Q increases, which

TABLE III
F-ratio COMPARISONS BETWEEN KD-RRT AND RRT FOR SIMULATIONS

Dim.	Failure rate(θ)	Block				Strip				Point			
		F-ratio(%)		Statistical analysis		F-ratio(%)		Statistical analysis		F-ratio(%)		Statistical analysis	
		RRT	KD-RRT	p-value	Effect size	RRT	KD-RRT	p-value	Effect size	RRT	KD-RRT	p-value	Effect size
	0.01	59.89	59.78	0.8369	0.4985	59.89	59.78	0.8369	0.4985	94.56	96.84	0.4333	0.4942
	0.005	58.98	61.35	0.0608	0.4860	58.98	61.35	0.0608	0.4860	94.59	93.87	0.5779	0.5041
	0.002	60.03	59.14	0.1020	0.5122	60.03	59.14	0.1020	0.5122	96.46	94.99	0.7895	0.4980
$d = 1$	0.001	59.65	58.81	0.3940	0.5064	59.65	58.81	0.3940	0.5064	99.12	98.31	0.9471	0.4995
	0.0005	58.03	59.88	0.2081	0.4906	58.03	59.88	0.2081	0.4906	92.98	96.17	0.4141	0.4939
	0.0002	59.64	58.57	0.1543	0.5106	59.64	58.57	0.1543	0.5106	94.26	97.71	0.2130	0.4907
	0.0001	60.19	59.78	0.7151	0.5027	60.19	59.78	0.7151	0.5027	98.48	93.42	0.0846	0.5129
	0.01	70.98	70.20	0.5199	0.4952	93.57	89.69	0.0098	0.5193	99.29	99.64	0.6694	0.4968
	0.005	71.86	71.39	0.8569	0.4987	93.79	94.39	0.8427	0.5015	97.18	97.89	0.9564	0.4996
	0.002	69.63	68.33	0.6560	0.5033	94.19	94.15	0.8248	0.4983	97.50	100.07	0.1459	0.4892
$d = 2$	0.001	67.68	68.46	0.5780	0.4959	98.99	94.86	0.0686	0.5136	98.72	98.02	0.8639	0.4987
	0.0005	67.25	69.82	0.1650	0.4897	96.55	97.87	0.2900	0.4921	97.08	96.38	0.7951	0.4981
	0.0002	67.80	67.89	0.9782	0.5002	96.77	100.11	0.2458	0.4913	96.72	95.84	0.5243	0.5047
	0.0001	66.50	68.49	0.5070	0.4951	95.61	99.91	0.2776	0.4919	94.05	98.60	0.0816	0.4870
	0.01	83.26	83.35	0.8338	0.4984	96.15	99.33	0.0690	0.4864	103.37	99.11	0.6081	0.5038
	0.005	82.78	82.13	0.5393	0.5046	97.19	95.78	0.9655	0.5003	101.92	101.06	0.7361	0.5025
	0.002	78.98	80.57	0.2017	0.4905	97.97	99.35	0.3265	0.4927	98.49	101.64	0.2771	0.4919
$d = 3$	0.001	78.05	76.95	0.5621	0.5043	101.76	100.15	0.8175	0.5017	98.98	102.64	0.3608	0.4932
	0.0005	78.08	77.50	0.5628	0.5043	99.76	102.11	0.4928	0.4949	99.55	98.18	0.6002	0.5039
	0.0002	75.35	78.05	0.1073	0.4880	99.84	99.18	0.7840	0.5020	100.12	96.54	0.0752	0.5133
	0.0001	78.95	78.10	0.4844	0.5052	99.52	101.21	0.3847	0.4935	97.79	98.40	0.9544	0.5004
	0.01	93.39	95.45	0.6505	0.4966	97.61	97.74	0.8009	0.5019	107.61	107.04	0.2683	0.5083
	0.005	92.51	92.97	0.8458	0.4986	98.10	100.31	0.7737	0.4979	106.12	106.75	0.5268	0.5047
	0.002	90.74	88.87	0.2034	0.5095	97.37	102.56	0.0224	0.4830	103.31	104.94	0.7101	0.4972
$d = 4$	0.001	90.18	88.62	0.6618	0.4967	99.28	96.52	0.0688	0.5136	105.12	100.58	0.0300	0.5162
	0.0005	86.63	86.22	0.4621	0.5055	100.55	96.14	0.2191	0.5092	102.34	100.92	0.5268	0.5047
	0.0002	87.44	85.77	0.2276	0.5090	101.56	100.08	0.7064	0.5028	99.95	98.32	0.5435	0.5045
	0.0001	83.15	84.47	0.4597	0.4945	99.86	102.02	0.4241	0.4940	97.92	101.87	0.4193	0.4940
	0.01	99.20	100.02	0.9243	0.4993	101.73	98.97	0.5157	0.5048	109.10	111.98	0.2740	0.4918
	0.005	98.93	99.74	0.9484	0.5005	101.90	96.99	0.0621	0.5139	110.41	108.23	0.6739	0.5031
	0.002	98.41	99.15	0.8874	0.5011	97.67	99.14	0.3050	0.4924	111.72	107.77	0.2372	0.5088
$d = 5$	0.001	98.43	100.64	0.7669	0.4978	99.33	100.65	0.3143	0.4925	111.82	105.09	0.0110	0.5190
	0.0005	94.44	94.97	0.9277	0.5007	99.89	102.27	0.6454	0.4966	103.83	108.77	0.1653	0.4897
	0.0002	93.67	95.52	0.7956	0.4981	102.51	101.35	0.9100	0.5008	106.74	105.38	0.9562	0.5004
	0.0001	95.17	90.74	0.2152	0.5092	96.39	100.56	0.0657	0.4863	104.00	108.44	0.2726	0.4918
	0.01	106.95	108.08	0.9017	0.4991	99.05	100.17	0.6961	0.4971	113.07	114.24	0.8323	0.5016
	0.005	108.77	107.80	0.8178	0.4983	96.94	97.93	0.8194	0.4983	112.24	111.49	0.4746	0.5053
	0.002	106.21	110.41	0.1640	0.4896	99.97	101.75	0.8809	0.5011	112.52	112.80	0.6379	0.4965
$d = 6$	0.001	106.16	106.91	0.8501	0.5014	100.75	98.47	0.1715	0.5102	111.26	110.42	0.8954	0.4990
	0.0005	103.29	104.38	0.4669	0.4946	99.92	96.89	0.6957	0.5029	112.41	114.29	0.5229	0.4952
	0.0002	101.14	99.78	0.6233	0.5037	98.14	98.32	0.7147	0.5027	104.04	110.08	0.0700	0.4865
	0.0001	102.32	100.49	0.4793	0.5053	100.02	99.47	0.5090	0.4951	104.95	106.56	0.3590	0.4932
	0.01	116.68	117.33	0.7586	0.5023	102.28	101.46	0.1781	0.5100	115.34	115.52	0.8426	0.4985
	0.005	118.92	115.79	0.2983	0.5078	101.27	97.69	0.9587	0.5004	115.54	117.21	0.5566	0.4956
	0.002	117.12	117.65	0.6216	0.4963	98.94	99.75	0.9056	0.5009	118.55	117.68	0.7849	0.4980
$d = 8$	0.001	112.48	118.67	0.0542	0.4856	101.33	98.55	0.3510	0.5070	115.27	114.41	0.9957	0.5000
	0.0005	111.89	115.90	0.3310	0.4928	100.49	100.58	0.6662	0.5032	115.07	122.04	0.0816	0.4870
	0.0002	110.91	112.76	0.5868	0.4959	102.04	99.08	0.3150	0.5075	114.11	116.35	0.7744	0.4979
	0.0001	113.47	115.94	0.4943	0.4949	103.34	99.67	0.1358	0.5111	114.08	116.57	0.1748	0.4899
	0.01	112.99	119.34	0.1164	0.4883	101.69	102.17	0.7343	0.4975	112.48	107.61	0.0722	0.5134
	0.005	120.21	119.19	0.6579	0.5033	100.54	99.09	0.3308	0.5072	113.95	116.27	0.6039	0.4961
	0.002	122.29	123.54	0.8742	0.4988	102.61	102.12	0.5572	0.5044	115.40	117.85	0.6512	0.4966
$d = 10$	0.001	119.37	120.96	0.1548	0.4894	100.45	101.20	0.3708	0.4933	120.41	119.12	0.5860	0.5041
	0.0005	118.23	120.04	0.7255	0.5026	99.76	99.94	0.5244	0.5047	124.51	117.84	0.0513	0.5145
	0.0002	118.60	120.16	0.8626	0.4987	99.37	99.57	0.9989	0.5000	125.05	121.77	0.5092	0.5049
	0.0001	117.89	121.82	0.1860	0.4901	100.51	97.17	0.3075	0.5076	119.03	122.84	0.1430	0.4891

means that more tree branches need to be backtracked, and thus the efficiency of the algorithm decreases significantly. In addition, inserting the test cases into the KD-Tree requires additional time overhead.

Table V reports average F-time comparisons between RRT and KD-RRT for all real-life programs. The F-time of KD-RRT is smaller than that of RRT regardless of the program. From the statistical analysis, all p-values are 0 and all effect sizes are greater than 0.5. This indicates that KD-RRT is quite different and significantly better than RRT.

In general, KD-RRT takes much less time than RRT, both for simulations and empirical studies. Especially when the

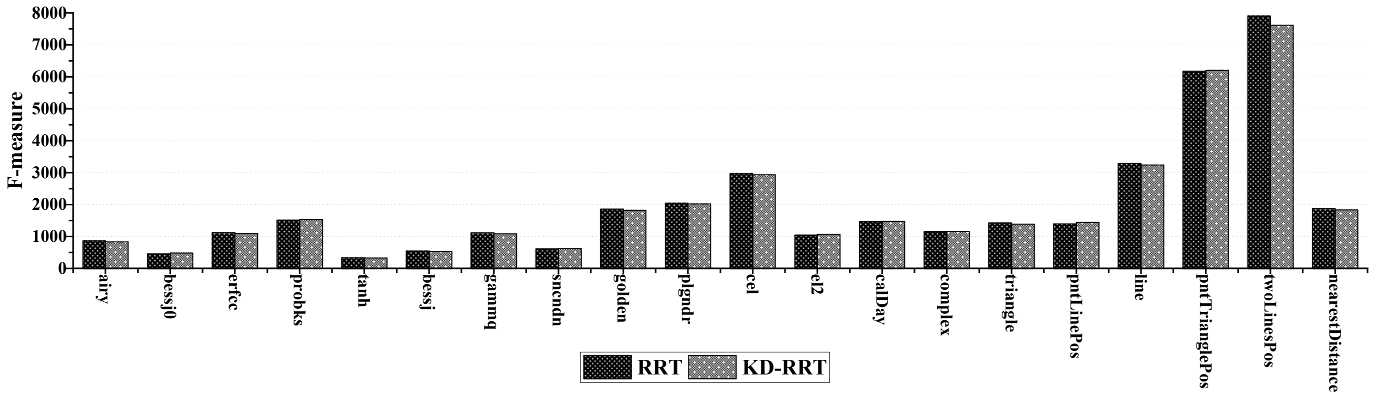


Fig. 4. F-measure comparisons between RRT and KD-RRT for object programs.

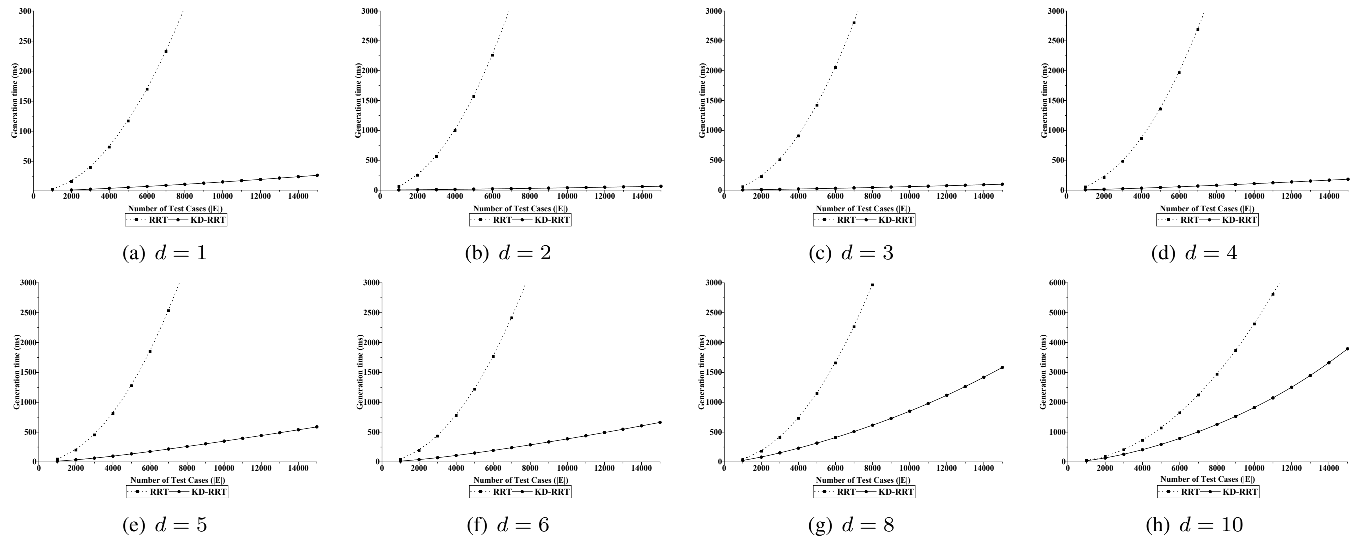


Fig. 5. Comparison of the time required to generate a set number of test cases between KD-RRT and RRT for different dimensions

dimension is low and a large number of test cases need to be generated.

To conclude, our KD-RRT approach performs similarly to RRT in terms of effectiveness, but is significantly more efficient. Therefore KD-RRT is more cost effective.

VI. Related Work

In this paper, we reduce the time overhead of RRT by using KD-Tree to compute the nearest neighbors. Here we briefly review some techniques related to this work.

Three *forgetting* strategies, called *Random Forgetting*, *Continuous Retention* and *Restarting*, were proposed by Chen et al. [47] to solve the problem of time overhead of RRT. However, their failure-detection capability is worse than RRT. And it is difficult to determine the value of the *Memory Parameter* k in the methods. KD-RRT also has the idea of "forgetting", but by using an efficient spatial index tree to "forget" executed test cases that are far away from the candidate test case. Mao et al. [18] proposed *KD-Tree-enhanced Fixed-size-Candidate-set ART* (KDFC-ART) to improve the efficiency of *Fixed-size-Candidate-set ART* (FSCS-ART). Our work extends the application of KD-Tree to RRT. This shows that KD-Tree can be widely used in ART to solve the problem of time overhead. Since KDFC-ART and KD-RRT are modifications of different methods, we have not included it in our experimental studies. Chen et al. [20] proposed *RRT by largest available zone* (RRT-LAZ) that proactively generates test cases outside the exclusion zone. The time complexity of RRT-LAZ is $O(n^2)$. RRT-LAZ involves partitioning, which is a different direction from KD-RRT. Some ART algorithms for non-numerical inputs have also been proposed. The linear-time ARTsum proposed by Barus et al. [48] uses the concepts of categories and choices from category-partition testing. The OMISS proposed by Chen et al. [49] uses a new similarity metric for object-oriented software(OOS) testing. The KD-RRT proposed in this paper focuses on programs with numerical inputs.

VII. Conclusion and Future Work

Adaptive Random Testing (ART) [9] improves the failure-detection capability of Random Testing (RT) [1], [2] by making the randomly generated test cases as evenly distributed as possible in the input domain. Restricted Random Testing

TABLE IV
Generation time COMPARISONS BETWEEN KD-RRT AND RRT FOR SIMULATIONS.

Dim.	Number of Tset Cases (n)	generation time(ms)		Statistical analysis	
		RRT	KD-RRT	p-value	Effect size
$d = 1$	1000	3.44	0.93	0.0000	0.9982
	2000	16.33	2.11	0.0000	1.0000
	5000	117.04	6.48	0.0000	1.0000
	10000	484.17	15.54	0.0000	1.0000
	15000	1165.13	26.59	0.0000	1.0000
$d = 2$	1000	61.95	2.36	0.0000	1.0000
	2000	249.14	5.35	0.0000	1.0000
	5000	1565.61	15.94	0.0000	1.0000
	10000	6341.41	37.87	0.0000	1.0000
	15000	14523.52	64.31	0.0000	1.0000
$d = 3$	1000	55.49	3.33	0.0000	1.0000
	2000	224.90	7.82	0.0000	1.0000
	5000	1422.19	23.92	0.0000	1.0000
	10000	5774.59	57.66	0.0000	1.0000
	15000	13261.12	97.82	0.0000	1.0000
$d = 4$	1000	52.51	5.44	0.0000	1.0000
	2000	212.84	13.38	0.0000	1.0000
	5000	1358.30	43.32	0.0000	1.0000
	10000	5579.87	107.52	0.0000	1.0000
	15000	12901.20	181.77	0.0000	1.0000
$d = 5$	1000	49.11	13.09	0.0000	0.9995
	2000	199.68	35.20	0.0000	1.0000
	5000	1276.95	134.49	0.0000	1.0000
	10000	5254.89	348.89	0.0000	1.0000
	15000	12145.32	587.36	0.0000	1.0000
$d = 6$	1000	47.13	13.81	0.0000	1.0000
	2000	191.50	38.87	0.0000	1.0000
	5000	1219.90	149.23	0.0000	1.0000
	10000	4983.03	385.97	0.0000	1.0000
	15000	11413.49	662.81	0.0000	1.0000
$d = 8$	1000	44.82	25.54	0.0000	1.0000
	2000	181.71	80.59	0.0000	1.0000
	5000	1147.85	315.33	0.0000	1.0000
	10000	4661.96	849.28	0.0000	1.0000
	15000	10653.15	1582.92	0.0000	1.0000
$d = 10$	1000	44.96	38.64	0.0000	0.9955
	2000	181.21	130.88	0.0000	1.0000
	5000	1137.97	585.24	0.0000	1.0000
	10000	4624.56	1820.20	0.0000	1.0000
	15000	10689.22	3789.26	0.0000	1.0000

TABLE V
F-time COMPARISONS BETWEEN RRT AND KD-RRT FOR OBJECT PROGRAMS.

Program	F-time(ms)		Statistical analysis	
	RRT	KD-RRT	p-value	Effect size
airy	4.46	1.00	0.0000	0.6923
bessj0	1.21	0.52	0.0000	0.5919
erfcc	7.78	1.36	0.0000	0.7215
probks	37.19	15.64	0.0000	0.6409
tanh	0.98	0.50	0.0000	0.5833
bessj	35.83	2.54	0.0000	0.7716
gammq	156.63	4.74	0.0000	0.8241
sncndn	50.74	1.58	0.0000	0.8165
golden	422.54	16.16	0.0000	0.8130
plgndr	396.22	30.00	0.0000	0.7657
cel	664.02	48.04	0.0000	0.7544
el2	120.63	8.44	0.0000	0.7573
calDay	218.52	37.37	0.0000	0.6606
complex	141.21	23.48	0.0000	0.6627
triangle	204.96	29.96	0.0000	0.6692
pntLinePos	198.70	30.75	0.0000	0.6668
line	1080.24	231.48	0.0000	0.6347
pntTrianglePos	3826.90	623.24	0.0000	0.6596
twoLinesPos	5859.36	828.61	0.0000	0.6905
nearestDistance	300.40	162.74	0.0002	0.5276

(RRT) [21], the classic algorithm for ART, achieves an even distribution of test cases by defining exclusion regions around each test case that has been executed but no failures have been found. However, one obvious shortcoming of RRT is that it requires a lot of distance calculations when selecting the next test case, resulting in a high time overhead. Therefore, this paper proposes an approach using KD-Tree [23], called KD-RRT. KD-RRT calculates the nearest neighbors of candidate test cases more intelligently instead of sequential distance calculations. KD-RRT uses the idea of pruning to ignore the distance calculations of test cases that are far away from the candidate test cases. At the same time, whenever a temporary nearest neighbor smaller than the exclusion radius appears in the process of finding the nearest neighbor, the finding is terminated and the candidate test cases are directly regenerated. Simulations and empirical studies on 20 real-life programs show that KD-RRT can significantly reduce the time overhead of RRT, while preserving the effectiveness of RRT.

In addition, KD-RRT suffers from the shortcomings of KD-Tree, and its efficiency in high dimensions is not optimistic. As future work, we will use the improved KD-Tree to calculate the nearest neighbor. And we will use KD-RRT to test larger-scale projects.

ACKNOWLEDGMENT

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

REFERENCES

[1] V. D. Agrawal, "When to use random testing," *IEEE Trans. Computers*, vol. 27, no. 11, pp. 1054–1055, 1978.

[2] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[4] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for java jit compiler test system," in *Third International Conference on Quality Software, 2003. Proceedings.* IEEE, 2003, pp. 20–23.

[5] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th USENIX Windows System Symposium*, vol. 4. Seattle, 2000, pp. 59–68.

[6] J. Regehr, "Random testing of interrupt-driven software," in *Proceedings of the 5th ACM International Conference on Embedded software*, 2005, pp. 290–298.

[7] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 1243–1251.

[8] W. Muangsiri and S. Takada, "Random gui testing of android application using behavioral model," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 09n10, pp. 1603–1612, 2017.

[9] T. Y. Chen, H. Leung, and I. Mak, “Adaptive random testing,” in *Annual Asian Computing Science Conference*. Springer, 2004, pp. 320–329.

[10] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, “A survey on adaptive random testing,” *IEEE Transactions on Software Engineering*, 2019.

[11] L. J. White and E. I. Cohen, “A domain strategy for computer program testing,” *IEEE transactions on software engineering*, no. 3, pp. 247–257, 1980.

[12] P. E. Ammann and J. C. Knight, “Data diversity: An approach to software fault tolerance,” *Ieee transactions on computers*, vol. 37, no. 4, pp. 418–425, 1988.

[13] G. B. Finelli, “Nasa software failure characterization experiments,” *Reliability Engineering & System Safety*, vol. 32, no. 1-2, pp. 155–169, 1991.

[14] P. Bishop, “The variation of software survival time for different operational input profiles. fault-tolerant computing, 1993. ftcs-23,” in *Digest of Papers., The Twenty-Third International Symposium on*, 1993, pp. 98–107.

[15] C. Schneckenburger and J. Mayer, “Towards the determination of typical failure patterns,” in *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 90–93.

[16] R. Huang, H. Liu, X. Xie, and J. Chen, “Enhancing mirror adaptive random testing through dynamic partitioning,” *Information and Software Technology*, vol. 67, pp. 13–29, 2015.

[17] 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, no. 9, p. 092106, 2017.

[18] C. Mao, X. Zhan, T. Tse, and T. Y. Chen, “Kdfc-art: a kd-tree approach to enhancing fixed-size-candidate-set adaptive random testing,” *IEEE Transactions on Reliability*, vol. 68, no. 4, pp. 1444–1469, 2019.

[19] H. Ackah-Arthur, J. Chen, D. Towey, M. Omari, J. Xi, and R. Huang, “One-domain-one-input: Adaptive random testing by orthogonal recursive bisection with restriction,” *IEEE Transactions on Reliability*, vol. 68, no. 4, pp. 1404–1428, 2019.

[20] J. Chen, Q. Bao, T. Tse, T. Y. Chen, J. Xi, C. Mao, M. Yu, and R. Huang, “Exploiting the largest available zone: A proactive approach to adaptive random testing by exclusion,” *IEEE Access*, vol. 8, pp. 52 475–52 488, 2020.

[21] K. P. Chan, T. Y. Chen, and D. Towey, “Restricted random testing,” in *European Conference on Software Quality*. Springer, 2002, pp. 321–330.

[22] J. Mayer and C. Schneckenburger, “An empirical analysis and comparison of random testing techniques,” in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, pp. 105–114.

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

[24] E. J. Weyuker, “On testing non-testable programs,” *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[25] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu, “Proportional sampling strategy: guidelines for software testing practitioners,” *Information and Software Technology*, vol. 38, no. 12, pp. 775–782, 1996.

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

[27] 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 (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[28] D. A. White and R. C. Jain, “Similarity indexing: Algorithms and performance,” in *Storage and Retrieval for Still Image and Video Databases IV*, vol. 2670. International Society for Optics and Photonics, 1996, pp. 62–73.

[29] R. F. Sproull, “Refinements to nearest-neighbor searching in k-dimensional trees,” *Algorithmica*, vol. 6, no. 1, pp. 579–589, 1991.

[30] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter, “Bkd-tree: A dynamic scalable kd-tree,” in *International Symposium on Spatial and Temporal Databases*. Springer, 2003, pp. 46–65.

[31] P. Ram and K. Sinha, “Revisiting kd-tree for nearest neighbor search,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 1378–1388.

[32] K. Zhou, Q. Hou, R. Wang, and B. Guo, “Real-time kd-tree construction on graphics hardware,” *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, pp. 1–11, 2008.

[33] W. Hou, D. Li, C. Xu, H. Zhang, and T. Li, “An advanced k nearest neighbor classification algorithm based on kd-tree,” in *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*. IEEE, 2018, pp. 902–905.

[34] M. Hughey and M. W. Berry, “Improved query matching using kd-trees: a latent semantic indexing enhancement,” *Information Retrieval*, vol. 2, no. 4, pp. 287–302, 2000.

[35] N. Bhatia *et al.*, “Survey of nearest neighbor techniques,” *arXiv preprint arXiv:1007.0085*, 2010.

[36] A. Moore, “An introductory tutorial on kd-trees,” in *IEEE Colloquium on Quantum Computing: Theory, Applications & Implications*, 1991.

[37] M. Yan, L. Wang, and A. Fei, “Artdl: Adaptive random testing for deep learning systems,” *IEEE Access*, vol. 8, pp. 3055–3064, 2019.

[38] A. Arcuri and L. Briand, “A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[39] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics*, vol. 1, no. 6, pp. 80–83, 1945.

[40] 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, pp. 163–183, 2012.

[41] ACM, “Collected algorithms of the acm,” [EB/OL], http://calgo.acm.org// Accessed May 20, 2021.

[42] W. H. Press, H. William, S. A. Teukolsky, W. T. Vetterling, A. Saul, and B. P. Flannery, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[43] 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, pp. 1331–1362, 2012.

[44] Y. D. Liang, *Introduction to Java programming and data structures*. Pearson Education, 2018.

[45] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

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

[47] K. P. Chan, T. Chen, and D. Towey, “Forgetting test cases,” in *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*, vol. 1. IEEE, 2006, pp. 485–494.

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

[49] J. Chen, F.-C. Kuo, T. Y. Chen, D. Towey, C. Su, and R. Huang, “A similarity metric for the inputs of oo programs and its application in adaptive random testing,” *IEEE Transactions on Reliability*, vol. 66, no. 2, pp. 373–402, 2016.