The Journal of Systems & Software 180 (2021) 111008

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

SWFC-ART: A cost-effective approach for Fixed-Size-Candidate-Set Adaptive Random Testing through small world graphs☆

Muhammad Ashfaq [a], Rubing Huang [a,b,*], Dave Towey [c], Michael Omari [d], Dmitry Yashunin [e], Patrick Kwaku Kudjo [f], Tao Zhang [b]

[a] School of Computer Science and Communication Engineering, Jiangsu University, Jiangsu 212013, China
[b] Faculty of Information Technology, Macau University of Science and Technology, Macau 999078, China
[c] School of Computer Science, University of Nottingham Ningbo, Zhejiang 315100, China
[d] Department of Computer Science, Takoradi Technical University, Western Region 23321, Ghana
[e] Harman X, Nizhny Novgorod Oblast 603105, Russia
[f] University of Professional Studies, Greater Accra Region 00233, Ghana

ARTICLE INFO

Article history:
Received 26 August 2020
Received in revised form 5 May 2021
Accepted 18 May 2021
Available online 27 May 2021

Keywords:
Software testing
Random testing
Adaptive random testing
Efficiency
Hierarchical Navigable Small World Graphs

ABSTRACT

Adaptive random testing (ART) improves the failure-detection effectiveness of random testing by leveraging properties of the clustering of failure-causing inputs of most faulty programs: ART uses a sampling mechanism that evenly spreads test cases within a software's input domain. The widely-used Fixed-Sized-Candidate-Set ART (FSCS-ART) sampling strategy faces a quadratic time cost, which worsens as the dimensionality of the software input domain increases. In this paper, we propose an approach based on small world graphs that can enhance the computational efficiency of FSCS-ART: SWFC-ART. To efficiently perform nearest neighbor queries for candidate test cases, SWFC-ART incrementally constructs a hierarchical navigable small world graph for previously executed, non-failure-causing test cases. Moreover, SWFC-ART has shown consistency in programs with high dimensional input domains. Our simulation and empirical studies show that SWFC-ART reduces the computational overhead of FSCS-ART from quadratic to log-linear order while maintaining the failure-detection effectiveness of FSCS-ART, and remaining consistent in high dimensional input domains. We recommend using SWFC-ART in practical software testing scenarios, where real-life programs often have high dimensional input domains and low failure rates.

1. Introduction

Software testing (Myers, 1979) is a fundamental software quality assurance activity. Random testing (RT) (Duran and Ntafos, 1981) is a popular black-box software testing technique that randomly selects and executes a subset of test cases from the software's input domain. Benefits of using RT include that it does not require explicit information, other than the inputs it takes, about the software under test (Agrawal, 1978), and that it can easily be automated (Hamlet and Maciniak, 1994). RT has been used to test many real-life software packages, including: Windows NT applications (Forrester and Miller, 2000); embedded software systems (Regehr, 2005); database systems (Bati et al., 2007; Slutz, 1998); Android applications (Muangsiri and Takada, 2017); Java Just-In-Time Compilers (Yoshikawa et al., 2003); .NET error detection (Pacheco et al., 2008); security assessment (Godefroid et al., 2008); Mac OS robustness assessment (Miller et al., 2006); graphical user interfaces (Daboczi et al., 2003); and UNIX utility programs (Miller et al., 1990, 1995).

Although RT has been reported to be effective, a large body of related research has highlighted continuing questions about its actual effectiveness (Myers, 1979; Duran and Ntafos, 1984). For example, RT does not take advantage of non-failure-revealing test cases, which may still have information about program behavior, and should not be discarded without careful inspection (Myers, 1979). Moreover, RT may not achieve satisfactory failure-detection effectiveness and code coverage (Chen et al., 2013), which may make it unsuitable when there are constraints on the number of test cases that can be executed.

An important finding in software testing has been the fact that inputs that cause programs to fail (failure-causing inputs), more often than not, form contiguous regions within the input

☆ Editor: Antonia Bertolino.

* Corresponding author at: Faculty of Information Technology, Macau University of Science and Technology, Macau 999078, China.
E-mail addresses: 5102180325@stmail.ujs.edu.cn (M. Ashfaq), rbhuang@ujs.edu.cn, rbhuang@must.edu.mo (R. Huang), dave.towey@nottingham.edu.cn (D. Towey), michael.omari@ttu.edu.gh (M. Omari), Dmitry.Yashunin@harman.com (D. Yashunin), kudjo@upsamail.edu.gh (P.K. Kudjo), tazhang@must.edu.mo (T. Zhang).

https://doi.org/10.1016/j.jss.2021.111008
0164-1212/

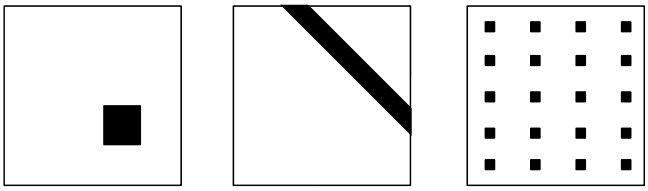

Fig. 1. Block, strip, and point failure patterns.

domain of a program (Schneckenburger and Mayer, 2007; Arcuri and Briand, 2011; Hamlet, 2002; Bishop, 1993; Chan et al., 1996; Finelli, 1991; Ammann and Knight, 1988; White and Cohen, 1980). A family of testing techniques called Adaptive Random Testing (ART) (Chen et al., 2005; Huang et al., 2019; Chen et al., 2019) is based on the idea that, if RT's basic technique is slightly modified, such that test cases are more evenly distributed, the chances of encountering failure-causing inputs can be significantly increased. Three broad categories of patterns of these failure-causing inputs have been identified (Chan et al., 1996), as shown in Fig. 1: *block*, *strip*, and *point* patterns.

An objective of ART is to minimize the number of test cases executions needed before revealing program failure. Theoretical support for ART includes: not repeating testing where failure is unlikely to occur, covering every aspect of the program (coverage), and exploring as much variety (diversity) in inputs as possible (Chen et al., 2010; Ammann and Knight, 1988). ART has been shown to be more effective than RT in real-life software testing scenarios — such as resource-constrained testing (Mayer and Schneckenburger, 2006) and beta testing (Chen et al., 2007a) — achieving better code coverage (Chen et al., 2013, 2008; Bueno et al., 2014), and using fewer test cases to find failure (Chen et al., 2005). ART can require up to 50% fewer test cases to find the first failure than RT (Chen and Merkel, 2008). Recently, ART has been gaining traction as a viable approach for testing real-life programs and systems, such as: testing deep neural networks (Yan et al., 2020); detection of cross-site-scripting (XSS) attacks (Lv et al., 2019); exposing SQL database vulnerabilities (Zhang et al., 2019); and testing object-oriented programs (Ciupa et al., 2008).

Several ART approaches exist, based on different strategies and motivations (Huang et al., 2019; Chen et al., 2019). The *Fixed-Size-Candidate-Set version of ART* (FSCS-ART) was the first proposed, and is the most-widely-used ART strategy, best known for its simplicity and failure-detection effectiveness (Huang et al., 2019). FSCS-ART follows the ART principle that if a test case does not reveal a failure, then nearby inputs are also unlikely to do so. Subsequent test cases, therefore, should be selected *far away* from the previously executed, non-failure-causing test cases. FSCS-ART achieves the concept of *far away* through computing distances among test cases.

Unfortunately, FSCS-ART incurs a quadratic time complexity when generating test cases, due to the brute-force strategy for determining the nearest neighbors of *candidate test cases* (see Section 2.1) — the distance between each candidate test case and *all* executed test cases must be calculated before the nearest neighbor can be identified. Furthermore, this computational cost rises sharply as the number of program input parameters (dimensions of the input domain, D) increases. These two issues can be collectively referred to as the *double-tier efficiency problem* of FSCS-ART. Given that most real-world programs have high dimensional input domains (Lin et al., 2009) and low failure rates (Arcuri and Briand, 2011) — which means that many test case executions may be required before finding a failure — the FSCS-ART double-tier efficiency problem needs to be addressed.

The work reported in this paper addresses the FSCS-ART double-tier efficiency problem. By identifying FSCS-ART as an instance of the nearest neighbor search (NNS) problem, we hypothesize that a solution to the FSCS-ART efficiency problem may lie in addressing the NNS mechanism: If the NNS mechanism can be scalable, and consistent with dataset size and dimensions, it may alleviate the double-tier efficiency problem. Furthermore, *approximate* NNS (ANNS) should be able to significantly alleviate the computational overheads of distance calculations, especially in high dimensional input domains (Indyk and Motwani, 1998). In software testing, NNS has been used to find the most similar test cases in regression testing (Bin Ali et al., 2019), test case prioritization (TCP) (Miranda et al., 2018), and model-based testing (Cartaxo et al., 2011). It has also been used to find the most diverse (opposite to similar) test cases in ART (Mao et al., 2019) and software product lines (Devroey et al., 2016). On the other hand, ANNS has been successfully applied to enhance the efficiency in other areas of software testing, including TCP (Miranda et al., 2018), test suite reduction (Cruciani et al., 2019), and prediction of test flakiness (Bertolino et al., 2020).

In this paper, we introduce an approach based on *Hierarchical Navigable Small World Graphs* (HNSWGs), a technology that has outperformed *tree*, *hashing*, and other *graph-based* NNS strategies on a wide variety of datasets (Li et al., 2019; Aumüller et al., 2020; Ponomarenko et al., 2014). HNSWGs represent an excellent potential solution for solving the computational overheads and high dimensionality problem of FSCS-ART. HNSWGs are built on navigable small world graphs with a controllable hierarchy for approximate k-nearest neighbor searches (Malkov and Yashunin, 2020), making them suitable for alleviating the exhaustive distance computations burden of FSCS-ART. The proposed method, referred to as *FSCS-ART by Hierarchical Navigable Small World Graphs* (abbreviated as SWFC-ART), stores previously executed, non-failure-causing test cases in an HNSWG data structure that is efficient for NNS queries, especially for high-dimensional datasets. We evaluated SWFC-ART in a series of simulations and empirical studies, examining its efficiency and failure-detection effectiveness. The results show that our proposed method reduces the computational overhead of FSCS-ART from quadratic to *log-linear* order while maintaining the failure-detection effectiveness of FSCS-ART, and remaining consistent in high dimensional input domains.

The rest of the paper is organized as follows: Section 2 introduces some background information for NNS, FSCS-ART, the current state-of-the-art for FSCS-ART (KDFC-ART), and HNSWGs. The proposed method is explained in Section 3. Section 4 describes the experimental setup used to evaluate the proposed method. Section 5 provides the experimental results and discussion. Section 6 describes potential limitations and threats to the validity of our work. Related work is described in Section 7. The paper concludes with Section 8, which also discusses some future work.

2. Background

2.1. Fixed-Size-Candidate-Set Adaptive Random Testing

The Fixed-Size-Candidate-Set implementation of ART (FSCS-ART) uses two sets of test cases: the candidate set (C), containing k elements; and the executed set (E) that contains the previously executed, non-failure-causing test cases. Initially, both E and C are empty. The first test case is randomly[1] generated and executed. If a failure is not found, the executed test case is added to E, and k new candidate test cases are randomly generated and added to C. The nearest neighbor in set E for each $c_i \in C$ $(1 \leq i \leq k)$ is determined by calculating the distance between

[1] Generally, with a uniform probability distribution.

c_i and all elements in E. Finally, the best candidate test case (c_{best}) is selected as the candidate whose nearest neighbor is farthest away—following the *min–max* strategy of FSCS-ART. This process can be described mathematically as follows (where δ is a similarity measure for test cases):

$$\forall c \in C, \min_{e \in E} \delta(c_{best}, e) \geq \min_{e \in E} \delta(c, e) \quad (1)$$

The *Euclidean distance* is typically used as δ for numeric programs (Huang et al., 2019; Chen et al., 2005): FSCS-ART chooses the test case from C that is most distant from previously executed, non-failure-causing test cases.

A core part of the FSCS-ART algorithm is finding the nearest neighbors of candidate test cases. Once the nearest neighbor of each candidate is found, c_{best} can be determined in constant time. If a candidate test case ($c_i \in C$) is considered a query point, and the executed test case set (E) is the dataset, then the whole process becomes an instance of the NNS problem. The NNS problem has been extensively studied in computer science, including in areas such as geographic information systems, artificial intelligence, pattern recognition, clustering, and outlier detection (O'Neil, 2017).

Definition 1. The NNS problem can be formally defined as follows: Given a d-dimensional input domain D (also called *vector space* or *input space*), and a distance function $\delta : D \times D \rightarrow \mathbb{R}$, for a finite set $X = \{x_1, x_2, x_3, \ldots, x_n\}$, where $X \subset D$, an *effective probability search method* is needed to find the $x_i \in X$ which is closest to $q \in D$ (according to δ). Each $x_i \in X$ and $q \in D$ are d-dimensional vectors.

$$NNS(q) = arg \min_{x \in X} \delta(q, x) \quad (2)$$

For FSCS-ART, X is a set of executed test cases (E), x_i is one executed test case (also called *test input*), and q is a candidate test case. An *effective probability search method* is not guaranteed to identify the exact nearest neighbor for a given candidate. Although the original FSCS-ART uses *exact* NNS, it has been found that an ANNS can also be employed that maintains the FSCS-ART failure-detection effectiveness (Mao et al., 2019).

FSCS-ART uses a brute-force NNS (Arya and Mount, 1993) for each element $c \in C$, with the distances between c and *each* element of E calculated to find its nearest neighbor. The complexity of the brute-force NNS is $O(d \cdot n)$ (Indyk and Motwani, 1998), and because there are k elements in C, one iteration of FSCS-ART has a complexity of $O(k \cdot d \cdot n)$. As the algorithm iterates n times, the total time complexity becomes $O(n \cdot k \cdot d \cdot n) = O(k \cdot d \cdot n^2)$. This quadratic time complexity can take a prohibitive amount of time when testing programs with high dimensionality and low failure rates.

Due to its simplicity, failure-detection effectiveness and popularity for testing numeric, non-numeric and object-oriented programs (especially after the various distance metrics proposed in these domains by ART researchers), most studies of the application of ART in testing real-life software packages have used FSCS-ART. As noted by Huang et al. (Huang et al., 2019), of 15 studies employing ART to test software packages from different application domains, 14 used FSCS-ART. In spite of this, however, the extent of research into improving FSCS-ART is less than that for partition-based ART strategies, which are less often used in real-life testing scenarios (Chen et al., 2019).

2.2. State-of-the-art: KDFC-ART

A state-of-the-art FSCS-ART overhead reduction strategy called KDFC-ART (Mao et al., 2019) stores previously executed, non-failure-causing test cases in a tree-based data structure to perform efficient NNS. LimBal-KDFC — the most efficient of the three KDFC-ART variants — incorporates limited backtracking and a semi-balancing strategy to perform ANNS, and appears to effectively address high-dimensionality computational challenges. Its worst-case time complexity is $O(k \cdot d^2 \cdot n \log n)$ — where k is the candidate set size; n is the number of generated test cases; and d is the input domain's dimensionality).

Previous studies have shown that tree-based approaches can perform NNS in low dimensional ($d \leq 5$) input spaces with $O(\log n)$ complexity. However, in worst-case situations, this complexity can become $O(d \cdot n^{1-1/d})$ (Lee and Wong, 1977). LimBal-KDFC, a tree-based search method, is therefore expected also to suffer from the impact of this phenomenon (see Section 7.1.5 for details).

2.3. Hierarchical navigable small world graphs

Graph-based approaches map vectors of a dataset into a graph data structure, and perform greedy traversals to find the nearest neighbor of a query point (Malkov and Yashunin, 2020). These approaches have been shown to out-perform both tree-based and hashing-based techniques (Chávez and Tellez, 2010; Arya and Mount, 1993; Wang et al., 2015; Aoyama et al., 2011; Paredes, 2008; Hajebi et al., 2011; Wang and Li, 2012; Jiang et al., 2016). However, these techniques face power-law scaling of the number of steps with the size of the dataset, and may potentially get stuck in local minima (Caretta Cartozo and De Los Rios, 2009; Dong et al., 2011).

To solve this problem, researchers have studied the construction of *Small World Graphs* (SWGs) instead of regular connected graphs. The *small world* phenomenon is related to the *Milgram Experiment* (Milgram, 1967), which showed that most social entities are linked through a small number of connections (average of 6). Watts & Strogatz (Watts and Strogatz, 1998) showed that, due to their high clustering and small path lengths, some real-life networks, called small world networks, can lie between *regular* and *connected* networks. These networks use a few long-range links as well as regular short-range links. Short-range links provide local connectivity by joining nodes with their neighbors. Long-range links are responsible for global connectivity, joining more distant nodes (Mehlhorn and Schreiber, 2013). Kleinberg (Kleinberg, 2000a,b) showed that if long-range links are introduced with a probability $r^{-\alpha}$ — where r is the distance between two distant nodes, and α is a fixed clustering coefficient — then the number of steps needed to reach the target node by a greedy search scales down to poly-logarithmic order. The value of α can be set to the dimensionality of the vector space. Based on this idea, many NNS and ANNS algorithms have been developed (Lifshits and Zhang, 2009; Karbasi et al., 2015; Beaumont et al., 2007a,b) that have reduced the greedy routing complexity from power-law to poly-logarithmic scaling. Small world properties can be incorporated into a graph during its construction (Malkov and Ponomarenko, 2016) — this has been used by NNS and ANNS, showing small world properties (Malkov et al., 2012, 2014).

Hierarchical Navigable Small World Graphs (HNSWGs) (Malkov and Yashunin, 2020) aim to further reduce the complexity of SWGs. HNSWGs are constructed by separating links into different layers based on their length: This means that only a fixed number of the connections for each element are evaluated (independently of the graph size), which allows for logarithmic scaling. Each element is assigned a layer level l, which denotes the highest layer it can belong to. The NNS is initiated from the top layer (which has the longest links), and continues until a local minimum is reached at that layer. The search then goes to the next lower layer, proceeding from the local minimum found in the upper layer. This process continues until the bottom layer.

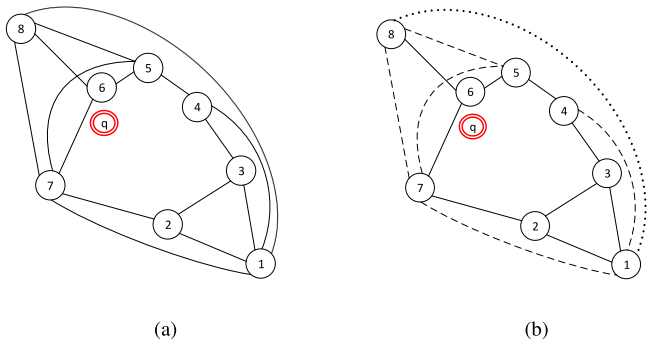

Fig. 2. HNSWG Structure: (a) Basic NSWG; (b) Break-down of links.

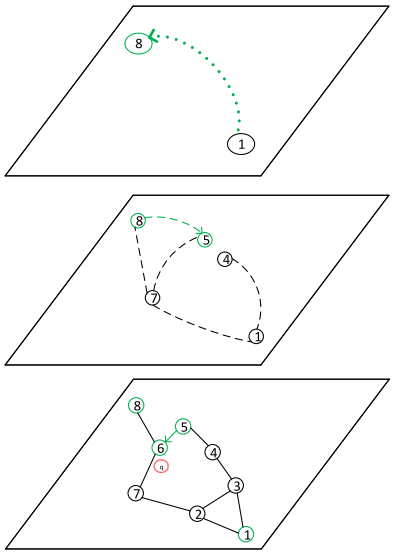

Fig. 3. Layered Hierarchy Representation of Fig. 2(b).

Because there is a fixed number of connections at each layer, if the layer level l is set with exponentially decaying probability, then the overall NNS complexity scales down to logarithmic order. The HNSWG structure is similar to probabilistic skip-lists (Pugh, 1990), with proximity graphs replacing linked-lists.

2.3.1. Example

Fig. 2(a) presents a sample *Navigable Small World Graph* (NSWG) where each node is connected to its neighboring nodes, and there are also some long-range links to more distant nodes. For example, Node 1 has bidirectional short-range links with its neighboring nodes (Nodes 2 & 3), and long-range links with Nodes 4, 7, and 8. This NSWG is converted to an HNSWG by grouping links into three categories: Long-, medium-, and short-range — which are represented in Fig. 2(b) by dotted, dashed, and solid lines, respectively. The long- and medium-range links — where a node is connected to nodes other than its neighbors — are responsible for the small world properties in the graph. The short-range links connect nodes to some of their neighbors, making an approximate Delaunay graph (Malkov et al., 2012, 2014). As shown in Fig. 3, links in the HNSWG are separated into three virtual layers (hierarchies) according to their length, with long-, medium-, and short-links categorized into the top, middle, and bottom layers, respectively.

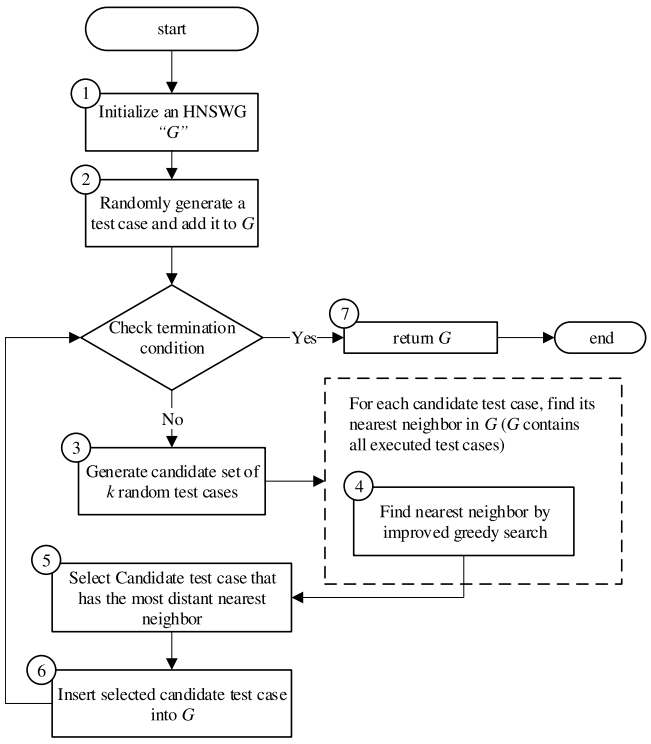

Fig. 4. SWFC-ART Framework.

2.3.2. NNS in HNSWG

An NNS for query point q (highlighted with a double circle in Figs. 2 & 3) starts from the top layer by selecting the node with the most links (the "maximum degree node") as an entry point (Node 1). The entry point is updated each time an element is inserted into the graph. The nearest neighbor of q in the top layer is determined (Node 8), and the search proceeds to the middle layer, restarting from the local minimum found in the top layer. The nearest neighbor is revised, with Node 5 now identified as the best possible solution. Finally, the search proceeds to the final (bottom) layer and again attempts to refine the nearest neighbors, resulting in Node 6 now being identified as the final nearest neighbor for q. According to this process, the nearest neighbor of q is found in three steps, compared with the eight distance calculations that would have had to be performed in a brute-force (exhaustive) search.

3. Method

This section introduces our proposed method, SWFC-ART (a **S**mall-**W**orld-graph-based approach for **F**ixed-size-**C**andidate-set **A**daptive **R**andom **T**esting), which uses an HNSWG to store E, and to efficiently find nearest neighbors for each $c_i \in C$.

3.1. Framework

The SWFC-ART method can be divided into seven major steps, as shown in Fig. 4. In the first step, an HNSWG G, which will map all executed test cases to its nodes, is initialized. The initialization phase requires a number of parameters (discussed in Section 3.3), including: the graph size, the number of nearest neighbors to be searched for in each layer, and the number of nearest neighbors to be connected for each inserted node. Once the graph is initialized, the first test case is randomly generated, inserted into the G (Step 2), and used to test the *software under test* (SUT). If the method

Algorithm 3.2.1 SWFC-ART

Input: 1. Size of candidate test case set: k
2. Program input domain: D
3. Distance function: δ
4. Nearest neighbors to be searched for at each layer: ef
5. Number of connections for each inserted element: M
6. Size of dynamic list for enhancing the accuracy of nearest neighbor search: $efConst$
7. Layers controller: m_l
8. Base size of the graph: b

Output: HNSWG object G consisting of executed test cases as its nodes

1: $d \leftarrow$ dimensionality of D
2: Globally-stored ep
3: Initialize G (d, δ, b, M, $efConst$, ef)
4: $t_{init} \leftarrow$ randomly generate a test case
5: Call `procedure G.Insert` (t_{init}, M, $efConst$, ep, m_l)
6: **while** termination condition is not satisfied **do**
7: $C \leftarrow$ Randomly generate k test cases
8: **for** each c in C **do**
9: Call `procedure G.NNS` (c, ef, ep)
10: **end for**
11: Select c_{best} from C
12: Call `procedure G.Insert` (c_{best}, M, $efConst$, ep, m_l)
13: **if** G.*currentSize*() **equals** b **then**
14: $b \leftarrow 2 \cdot b$
15: re-calculate $efConst$
16: $G_{temp} \leftarrow G$.*items*()
17: Re-initialize G (d, δ, b, M, $efConst$, ef)
18: $\forall t \in G_{temp}$, `G.Insert` (t, M, $efConst$, ep, m_l)
19: **end if**
20: **end while**
21: **return** G

has not terminated, Step 3 involves randomly generating k candidate test cases. In Step 4, the multilayered G is traversed to find the nearest neighbor for each candidate. Step 5 determines the best candidate (the one whose nearest neighbor is most distant). In Step 6, the best candidate is inserted into G, and used as the next test case for the SUT. This process repeats until a termination criterion is reached. Possible termination criteria include: finding a failure; executing a specific number of test cases; running the algorithm for a specific time limit; or any other specified criterion. If a termination criterion is satisfied, the algorithm terminates and returns G with all executed test cases as its nodes (Step 7).

3.2. SWFC-ART

SWFC-ART is a modified form of the FSCS-ART algorithm (Chen et al., 2005), storing previously executed, non-failure-causing test cases in an HNSWG (Malkov and Yashunin, 2020), instead of the arrays and trees used by FSCS-ART and LimBal-KDFC, respectively.

3.2.1. Algorithm

SWFC-ART takes eight inputs: (1) k (the size of the candidate test set); (2) D (the SUT's input domain); (3) δ (the distance function); (4) ef (the size of the dynamic list for the number of nearest neighbors to be searched for in each layer); (5) M (the number of connections for an inserted test case in each layer of the HNSWG); (6) $efConst$ (the size of the dynamic list for enhancing the accuracy of returned nearest neighbors — although this parameter is the same as ef, a different value is used during HNSW construction,

Procedure 3.2.2 `G.Insert` (t, M, $efConst$, ep, m_l)

1: $l \leftarrow \lfloor -\ln(unif(0...1)) \cdot m_l \rfloor$
2: $L \leftarrow$ level of ep // entry point to top layer
3: $W \leftarrow \phi$
Phase I
4: **for** $l_c \leftarrow L...l$ **do**
5: $W \leftarrow$ `procedure Searcher`(t, ep, $ef = 1$, l_c)
6: **end for**
Phase II
7: **for** $l_c \leftarrow l...2, 1, 0$ **do**
8: $W \leftarrow$ `procedure Searcher`(t, ep, $efConst$, l_c)
9: bidirectionally connect M (M_0 if $l_c = 0$) elements from W to t
10: shrink connections if needed
11: **end for**
12: **if** $l > maxLayer$ **then**
13: $ep \leftarrow t$ // update entry point
14: $maxLayer \leftarrow l$ // update maximum layer level
15: **end if**
16: **return** updated G with inserted t

and therefore we call it $efConst$ (efConstruction); (7) m_l (a non-zero integer to control the number of layers with exponentially decaying probability); and (8) b (the initial base size of the graph, representing the number of nodes that it can accommodate). The parameters are discussed further in Section 3.3. The algorithm returns an HNSWG (G) whose nodes are the executed test cases, the number of which corresponds to the *F-measure* (Section 4.2).

The algorithm begins by calculating the dimensionality d of the input domain D (line 1). The entry point ep (line 2) is stored globally, and updated each time an element is inserted into G — this differentiates HNSWGs from NSWGs, where the entry point is randomly chosen on each search iteration. On line 3, G has been initialized by specifying the parameter values. In the next phase, a randomly-generated test case from the SUT's input domain is executed, and inserted into G (lines 4–5). If no termination criterion has been satisfied, then k test cases are randomly generated, and put into C (line 7). The nearest neighbor of each candidate ($c \in C$) is determined by calling the `NNS procedure` (Section 3.2.3), and the candidate with the maximum distance from its nearest neighbor (c_{best}) is selected as the next test case (line 11). This selected test case is inserted into G by calling the `Insert procedure` (Section 3.2.2). The `if block` (lines 13–19) maintains the dynamic size of G, which is doubled if the number of stored test cases reaches the limit (b) (causing $efConst$ to be re-calculated).

The `NNS` and `Insert` procedures called by SWFC-ART require the `Searcher` procedure to identify the nearest neighbors on each layer of the HNSWG. Because these procedures have been comprehensively explained by Malkov & Yashunin (Malkov and Yashunin, 2020), the following is only a general overview.

3.2.2. Insert procedure

The `Insert procedure` takes a test case t, entry point ep, and three integer value parameters ($efConst$, M and m_l), and returns an updated G reflecting the insertion of t.

For each t, a maximum layer l is randomly selected with an exponentially decaying probability distribution (normalized by m_L) (line 1). L represents the layer of entry point node ep, which is the top layer of G. W, which is initially empty, stores the nearest neighbors of t in each layer (line 3). The NNS for t consists of two phases: In Phase I (lines 4–6), the search moves from the top layer L, to t's layer l, identifying exactly *one* nearest neighbor in each

Procedure 3.2.3 `G.NNS` (t, ef, ep)

1: $L \leftarrow$ level of ep // entry point to top layer
2: $W \leftarrow \phi$ // List of currently found nearest neighbors
Phase I
3: **for** $l_c \leftarrow L \ldots 1$ **do**
4: $W \leftarrow$ **procedure** `Searcher`(t, ep, $ef = 1$, l_c)
5: **end for**
Phase II
6: $W \leftarrow$ **procedure** `Searcher`(t, ep, ef, l_0)
7: Sort W in ascending order
8: **return** first element of W as nearest neighbor of t

Procedure 3.2.4 `Searcher` (t, ep, ef, l)

1: $d_{init} \leftarrow \delta(t, ep)$ // initial distance from entry point to test case
2: $W \leftarrow ep$
3: **for** each $e \in neighborhood(ep)$ at layer l **do**
4: **if** $\delta(t, e) < d_{init}$ **then**
5: $W \leftarrow W \bigcup e$;
6: greedily search neighborhood of e recursively
7: **end if**
8: **end for**
9: sort W in ascending order
10: **return** first ef elements of W as closest neighbors of t at layer l

layer. In Phase II (lines 7–11), each layer below l is searched for *efConst* nearest neighbors, with a goal of increasing the accuracy of the greedy search at lower layers (line 8). The best M nodes from W are linked to t. If the current layer is the ground layer (l_0), then t is linked to M_0 neighboring nodes (Section 3.3.5). The maximum number of connections for an element is kept to within a fixed limit, thus maintaining logarithmic complexity (line 10). The entry point and maximum layer level are updated (lines 12–15), and the updated G is returned (line 16).

3.2.3. NNS procedure

The `NNS procedure` is similar to the `Insert procedure`, with the exception that Phase I spans from the top layer to the second last layer. Similar to `Insert`, exactly *one* nearest neighbor is identified for t in Phase I (line 3–5). In Phase II, only the bottom layer (l_0) is searched for ef nearest neighbors (line 6). Finally, W is sorted according to $\delta(W[i], t) < \delta(W[i+1], t)$, and the first element is returned as the nearest neighbor of t (line 8).

3.2.4. Searcher procedure

Both `NNS` and `Insert` procedures call the `Searcher procedure` (a type of greedy search) to search for ef nearest neighbors in a given layer. `Searcher` identifies the nearest neighbors of test case t in layer l, given entry point ep, and returns ef nearest neighbors. Initially, the entry points are taken as temporary nearest neighbors (line 1), and stored in W. Next, the neighborhood of each entry point is recursively searched, in a greedy manner, for other nearest neighbors (lines 3–8), with any identified closer neighbors added to W. The first ef elements of W (that are at a minimum distance from t) are then returned.

3.3. Parameter optimization

SWFC-ART is controlled by a number of parameters: k, ef, *efConst*, M, b and δ. If all the parameter values are set to the minimum possible, then the HNSWG is not used, and the algorithm's effectiveness may become similar to that of RT. Using the optimal parameter values, as explained in the following, is therefore critical to the success of SWFC-ART.

3.3.1. Number of candidate test cases (k)

In most ART studies, the size of the candidate test set (the number of test cases randomly generated in each iteration), k, is usually set to 10 (Chen et al., 2005).

3.3.2. Graph size (b)

The size of the HNSWG needs to be given in advance, with larger graphs incurring more construction time (lowering efficiency). However, while this parameter has no apparent impact on the failure-detection effectiveness, it can affect efficiency. The actual number of nodes (executed test cases) in the final graph corresponds to the F-measure (the number of test cases executed before finding the first failure), which can depend on the failure rate of the SUT (which is unknown in advance). In ART, test cases are usually generated incrementally until a termination criterion is reached. One approach to deal with this would be to assign the maximum available size (as supported by the hardware and software platform) to the HNSWG, but this can incur a very heavy construction cost, especially for software with high failure rates. Because our analysis of varying b between 10^2 and 2×10^7 showed that the graph construction time for 10^2 to 10^4 remained relatively stable, but then increased significantly for larger sizes, we initially set b to 10^4, and double this any time additional nodes are needed. In practice, testers may set b according to their own specific needs.

3.3.3. Distance function (δ)

Because the Euclidean distance has been used in many FSCS-ART studies (Chen et al., 2005; Huang et al., 2019), especially for numeric programs, we adopted it in our simulations and experiments.

3.3.4. Size of dynamic list (ef)

The size of the dynamic list (ef) controls the number of a candidate's closest neighbors that are searched for in layers higher than its own layer. Because SWFC-ART employs ANNS, the identified nearest neighbor may not be the *actual* nearest neighbor. There is a tension between the efficiency and effectiveness for the ef value: Increasing ef incurs an additional time cost, but also increases the chances of finding actual nearest neighbors. It should be at least equal to the number of desired nearest neighbors for a candidate, and, since ART seeks only one nearest neighbor for each candidate, the minimum value of ef can be 1. With $ef = 1$, our analysis showed NNS accuracy of 90% in all dimensions and failure rates under study. Increasing ef to 2 incurred a little additional time cost, but also raised the accuracy to 98%. Because the scope of our study was to increase efficiency while keeping effectiveness at a comparable level to the state-of-the-art, and $ef = 2$ shows effectiveness similar to that of FSCS-ART and LimBal-KDFC (Section 5), we did not increase ef beyond 2. Testers may choose to increase ef if they are interested in further enhancing the effectiveness.

3.3.5. Number of links (M)

This parameter controls the number of connections made to an inserted element: More connections increase the failure-detection effectiveness, but compromise the efficiency. Following Malkov et al. (Malkov et al., 2012), who recommended that a newly-inserted element should be connected to at least its $M = 3 \cdot d$ closest neighbors (where d is the dimensionality of dataset), we set $M = 3 \cdot d$. On the ground layer (l_0), a separate

parameter M_0 has been used. Setting $M_0 = M$ reduces the NNS accuracy. $M_0 = 2 \cdot M$ is the recommended choice, because higher values can lead to performance degradation and excessive memory usage (Malkov and Yashunin, 2020).

3.3.6. Construction parameter (efConst)

The construction parameter (*efConst*) specifies the number of nearest neighbor candidates used during graph construction. As only M closest candidates are connected to the inserted element, $efConst \geq M$. In the `Insert` procedure, a candidate has to be searched for only one nearest neighbor in layers above its own layer, but this is increased to *efConst* for lower layers to improve the NNS accuracy. The value of *efConst* is logarithmic to the size of the dataset and is very similar to the w parameter described by Malkov et al. (Malkov et al., 2014): $A \cdot \log(N)$ (where N is the size of the dataset, and A can be any natural number).

3.4. Illustration

Fig. 5 demonstrates an example of inserting two new nodes into an HNSWG. Without loss of generality, the parameter values in this example were set as follows: $ef = 1$; $M = 3$; $M_0 = 2 \cdot M = 6$; and $efConst = 3$. (The value of m_l is not important for demonstration purposes.) The initial HNSWG is shown in Fig. 5(a), with long-range links in the top layer (Layer 2), and short-range links in the ground layer (Layer 0). Node 4 is the entry point to the graph.

Fig. 5(b) shows the process of inserting Node 7 into the HNSWG. Suppose that Node 7 is inserted into Layer 1, which means that it will be inserted into this, and all lower layers. The search starts from the top layer (Layer 2), with Node 4 as the entry point. (As a reminder: For layers above the insertion layer, the search identifies the *ef* closest elements; and for the insertion and lower layers, the search identifies the *efConst* closest elements.) In Layer 2, all neighbors of Node 4 are examined, and the *one* ($ef = 1$) closest to the inserted node is identified — in this case, Node 4. Node 4 is then used as the entry point for Layer 1, where its neighbors (Nodes 2 and 5) are examined, and then their neighbors are examined. This continues until all the neighborhood has been examined. In this example, because $efConst = 3$, Nodes 2, 4, and 5, are identified as the closest elements, and are connected directly to Node 7. They will then be used as entry points to Layer 0. At this stage, because Node 4 now has four connections, which exceeds the limit ($M = 3$), its closest M neighbors are identified and any remaining connections are discarded: The connection between Nodes 4 and 6 is therefore removed (represented by the dotted blue line in Layer 1 of Fig. 5(b)). Finally, in Layer 0, the neighbors of the entry points (Nodes 2, 4, and 5) are examined, and the $efConst = 3$ closest elements are identified and connected to Node 7: Nodes 1, 2, and 4.

Fig. 5(c) shows the insertion of Node 8. Suppose that Layer 0 is the insertion layer for Node 8. The search again begins from the top layer (Layer 2), from the entry point Node 4. In Layer 2, all neighbors of Node 4 are examined, and the *one* ($ef = 1$) closest to the inserted node is identified — Node 6. Node 6 is then used as the entry point for Layer 1, where examination of the various connected neighbors results in Node 5 being identified as the *one* closest node to Node 8, and thus being used as the entry point to Layer 0. In Layer 0, the neighbors of Node 5 are examined, and the *three* ($efConst = 3$) closest elements are identified and connected to Node 8: Nodes 3, 5, and 6.

Fig. 5(d) shows the updated HNSWG after insertion of both Nodes 7 and 8. The `NNS Procedure` is similar to the `Insert Procedure`, with the slight differences that $ef = 1$ nearest neighbors are searched for in *all* layers except the ground layer (l_0), where $efConst = 3$ nearest neighbors are identified (as explained in Section 3.2.3); and that the node is not inserted into the HNSWG (bidirectional connecting does not take place).

3.5. Complexity analysis

The time complexity of SWFC-ART can be considered in two parts: the NNS; and the HNSWG construction complexity. For each candidate test case, the algorithm searches for *ef* or *efConst* nearest neighbors on each layer, of which there are a maximum of m_l in the graph). Because *ef*, *efConst*, and m_l are constant values (and thus do not depend on the size of the dataset), the overall NNS complexity scales down to $O(\log n)$ for one candidate test case, and $O(k \cdot \log n)$ for k candidates. The HNSWG is constructed through sequential insertion of test cases, for each of which M nearest neighbors are connected in each layer. The complexity of inserting one test case into the HNSWG therefore becomes equal to the search complexity $O(\log n)$. The total graph construction complexity for the sequential insertion of n test cases scales to $O(n \cdot \log n)$. For k candidates and a d-dimensional input domain the overall complexity becomes log-linear (also called *linearithmic* or *quasilinear*): $O(k \cdot d \cdot n \cdot \log n)$. The storage complexity of SWFC-ART depends on the number of links (both long- and short-range): For four billion elements (nodes or test cases), four-byte unsigned integers can be used to store the HNSWG connections. The typical memory requirement for one HNSWG object is about 60–450 bytes, which has been confirmed by simulation analysis (Malkov and Yashunin, 2020).

4. Experimental studies

Our study aimed at solving the double-tier efficiency problem of FSCS-ART, an ART version known for its failure-detection effectiveness and application in real-life programs. In addition to FSCS-ART, LimBal-KDFC (Section 2.2) was also selected as a baseline for comparison.

4.1. Research questions

The double-tier efficiency problem conceptualizes two efficiency issues of the FSCS-ART algorithm. The first issue relates to the growing executed test set size when failure has not yet been revealed: This is a *scalability* issue. The second issue relates to the computational load associated with dimensionality increases for any size of test set: This is a *consistency* issue. In addition to examining the effect of these issues, we also wanted to investigate the impact (similarities and differences) of the ANNS strategies of LimBal-KDFC and SWFC-ART on the failure-detection effectiveness of FSCS-ART. Therefore, the following research questions were designed to guide our experiments:

- **RQ1:** Does SWFC-ART successfully solve the double-tier efficiency problem? (Efficiency)
- **RQ2:** How effective is SWFC-ART at revealing failures? (Effectiveness)
- **RQ3:** How evenly does SWFC-ART distribute test cases? (Test case distribution)

4.2. Evaluation metrics

4.2.1. Efficiency metrics

Because a goal of this study was to reduce the computational cost associated with FSCS-ART generating test cases, the *test case generation time* (T_G) was adopted an efficiency metric. The *test case execution time* (T_E) was also recorded. T_G includes the time taken to generate a fixed number of test cases, with lower times indicating better efficiency; while T_E is the total time taken by a program to execute the generated test cases.

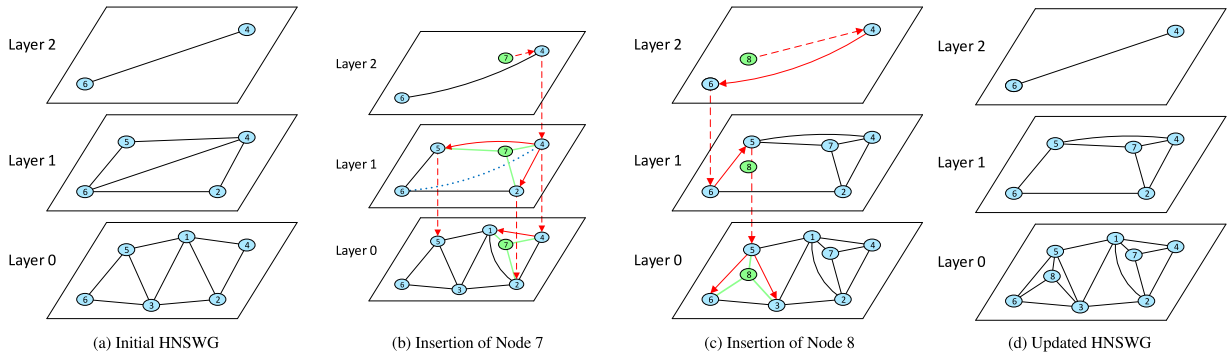

Fig. 5. Sample illustration of HNSWG node insertion.

4.2.2. Failure-detection effectiveness metrics

The *F-measure* is defined as the expected number of test case executions required by a method to find the first failure (Chen et al., 2005), with lower F-measure values (fewer test cases to find a failure) corresponding to better effectiveness. The F-measure was used as the failure-detection effectiveness metric in our study. If the failure rate, θ, of an SUT is defined as the ratio of the failure-causing inputs to the total size of the SUT input domain, then the theoretical *F-measure* of (F_{RT}) (with replacement) is $1/\theta$. Because ART aims to improve on the failure-detection effectiveness of RT, a measure of the extent of this improvement, known as the *F-ratio* (F_{ART}/F_{RT}), is also used in this paper.

4.2.3. Test case distribution metrics

Discrepancy refers to the differences of point densities in different sub-domains of the software input domain (D) — larger sub-domains should have more test cases than smaller ones. In an ideal situation, discrepancy values should be zero, indicating that the test cases (E) are evenly distributed. The input domain can have an infinite number of sub-domains (Liu et al., 2011); its Monte Carlo approximation can be obtained by (Chen et al., 2007b):

$$Discrepancy = \max_{i=1\cdots m} \left| \frac{|E_i|}{|E|} - \frac{|D_i|}{|D|} \right| \tag{3}$$

where D_1, D_2, D_3, $\cdots$, D_m are hyper-rectangular sub-domains of D whose size and location are randomly defined with uniform probability (Chow et al., 2013); E_1, E_2, E_3, $\cdots$, E_m are the subsets of E falling in each sub-domain, respectively; and m is the number of randomly defined sub-domains. A value of m that is too low causes unreliable approximation, but a value that is too high incurs significant overheads for discrepancy calculation (Ackah-Arthur et al., 2019) (1000 is a commonly-used value (Ackah-Arthur et al., 2019; Liu et al., 2010; Chen et al., 2007b)).

4.3. Simulations and subject programs

To answer RQ1, the T_G values of FSCS-ART, LimBal-KDFC, and SWFC-ART were recorded for test suites of sizes 500, 1000, 2000, 5000, 10,000, 15,000 and 20,000, in 2-, 3-, 4-, 5-, 10-, and 15-dimensional input domains. The T_G and T_E for the real-life programs (discussed below) were also recorded.

The failure-detection effectiveness of ART methods depends on several factors, including the shape of the failure region, the failure rate (θ), and the dimensionality (d) of the input domain (D) (Chen et al., 2007a). It is common practice in ART studies investigating failure-finding effectiveness according to the *F-measure* (F_{ART}) to use both simulations and empirical studies. We have followed this tradition in our study to answer RQ2.

Generally speaking, the failure-causing inputs of a software tend to cluster into *block*, *strip*, or *point* failure patterns (Chan et al., 1996). In our simulations, the block patterns were created by randomly generating a hyper-cube in D whose hyper-volume and length of each side equaled θ and $\sqrt[d]{\theta}$, respectively. The strip patterns were simulated by randomly selecting points on adjacent borders of D, joining them, and expanding the strip magnitude until the hyper-volume became equal to θ. Strips generated in corners of the input domain were discarded due to their unrealistic thickness. The point failure pattern was simulated by randomly generating 25 small, non-overlapping, block failure patterns, with the total hyper-volume of all the blocks being appropriate for the given θ. Simulations were performed for all three failure pattern types, with $d = \{2, 3, 4, 5, 10\}$ and $\theta = \{0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002, 0.0001\}$.

To answer RQ3, 100, 1000, and 10,000 test cases were generated in 2-, 3-, 4-, 5-, 10-, and 15-dimensional hyper-cube input domains using FSCS-ART, LimBal-KDFC, and SWFC-ART. Each dimension of the hyper-cubes was continuous, ranging from −5000 to 5000.

We also used 28 programs, of different sizes and dimensions, in our empirical studies. Faults[2] were seeded into the programs using the following mutation operators (Jia and Harman, 2010): constant replacement (CR); arithmetic operator replacement (AOR); return statement replacement (RSR); scalar variable replacement (SVR); statement deletion (SDL); and relational operator replacement (ROR). Table 1 summarizes details about their d, D, size (in terms of lines of code), fault types, number of seeded faults, and θ. (The "NA" for θ in some cases represents situations where the failure rate was not calculated.) The first 12 subject programs have been commonly used in ART research, and are from *Numerical Recipes* (Press et al., 2007) and *ACM Collected Algorithms* (Collected Algorithms of the ACM, 2020). The programs *calDay*, *complex* and *line* are from Ferrer et al. (Ferrer et al., 2012). The programs *pntLinePos*, *pntTrianglePos*, *twoLinesPos* and *triangle* were written as exercises from the textbook *Introduction to Java Programming and Data Structures* (Liang, 2017). The *nearestDistance* program takes five points in a 2-dimensional space and returns the two points that are nearest to each other (Mao et al., 2019). The *calGCD* program takes 10 integers and returns their greatest common divisor. The *select* program (May, 2007) returns the kth largest element from an unordered array. The *tcas* program is an aircraft collision avoidance system, from Siemens (Do et al., 2005). The *asinh*, *binomial*, and *period* programs are from Walkinshaw & Fraser (Walkinshaw and Fraser,

[2] According to the IEEE (ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary, 2010) a *fault* (*defect* or *bug*) is an oversight of a programmer. When *fault* is confronted during program execution, *failure* is said to have occurred i.e. software behaves unexpectedly.

Table 1
Details of the subject programs.

Program	d	Input Domain (D)		Size (LOC)	Fault types	Total faults	θ
		from	to				
bessj0	1	−300000	3000000	28	AOR,ROR,SVR,CR	5	0.001373
airy	1	−5000	5000	43	CR	1	0.000716
asinh	1	−10000	10000	360	AOR,ROR	2	0.0001001
erfcc	1	−30000	30000	14	AOR,ROR,SVR,CR	4	0.000574
probks	1	−50000	50000	22	AOR,ROR,SVR,CR	4	0.000387
tanh	1	−500	500	18	AOR,ROR,SVR,CR	4	0.001817
bessj	2	(2, −1000)	(300, 15000)	99	AOR,ROR,CR	4	0.001298
gammq	2	(0, 0)	(1700, 40)	106	ROR,CR	4	0.000830
sncndn	2	(−5000, −5000)	(5000, 5000)	64	SVR,CR	5	0.001623
binomial	2	(0, 0)	(128, 128)	501	CR	1	0.0001341
plgndr	3	(10, 0, 0)	(500, 11, 1)	36	AOR,ROR,CR	5	0.000368
golden	3	(−100, −100, −100)	(60, 60, 60)	80	ROR,SVR,CR	5	0.000550
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
period	4	(−10000, −10000, −10000, −10000)	(10000, 10000, 10000, 10000)	1128	CR	1	NA
calDay	5	(1, 1, 1, 1, 1800)	(12, 31, 12, 31, 2020)	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
matrixProcessor	12	(-10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4)	(10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4)	462	CR	1	NA
java.util.Arrays	15	(-10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4, -10^4)	(10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4, 10^4)	1357	CR	1	NA

2017). The program *matrixProcessor*[3] manipulates matrices according to specified matrix operations, and was written as an exercise from *JetBrains Academy* (JetBrains Academy, 2020). The *java.util.Arrays* (Java API Documentation, 2020; Giorgetti et al., 2010) is an array manipulation library in the Java API that contains various array helper functions (such as for searching and sorting).

4.4. Data collection and statistical analysis

The simulations were conducted by continuously generating test cases, using one of the testing strategies under study, until a test case fell *inside* the failure region. In the experiments with real programs, failure-revealing test inputs were identified when the output of the fault-seeded program differed from the output of the original. The number of test cases generated and executed before a failure was found (the F-measure) was recorded.

All trials were run S times to ensure that mean values had a 95% confidence level and 5% accuracy range, according to the central limit theorem (Freund, 1988; Chen et al., 2005). In the simulations, S was set to 10,000 for calculating the F-measure, and set to 1000 for T_G. In the empirical studies, S was also set to 10,000 trials. The sample sizes were confirmed to be large enough to obtain results with the desired confidence level and accuracy.

We used the unpaired two-tailed *Wilcoxon rank-sum test* (Wilcoxon, 1945) (reciprocal of *Mann–Whitney U test* (Mann

[3] https://github.com/ritish78/NumericMatrixProcessor.

Table 2
Wilcoxon Rank-Sum Tests and Effect Size Analyses of Test Case Generation Times for FSCS-ART, LimBal-KDFC and SWFC-ART.

d	n	Mean running time (ms)			FSCS-ART vs SWFC-ART		LimBal-KDFC vs SWFC-ART	
		FSCS-ART	LimBal-KDFC	SWFC-ART	p-value	effect size	p-value	effect size
2	500	21.86	**6.38**	28.81	0.0000	0.8573	0.0000	0.8658
	1000	99.60	**14.20**	57.24	0.0000	0.8658	0.0000	0.8658
	2000	406.98	**33.88**	127.23	0.0000	0.8658	0.0000	0.8658
	5000	2347.86	**92.00**	367.43	0.0000	0.8658	0.0000	0.8658
	10000	8466.74	**202.73**	818.66	0.0000	0.8658	0.0000	0.8658
	15000	18367.03	**328.03**	1525.58	0.0000	0.8658	0.0000	0.8658
	20000	32235.35	**470.50**	2145.45	0.0000	0.8658	0.0000	0.8658
3	500	26.15	**9.87**	30.37	0.0000	0.8535	0.0000	0.8658
	1000	113.63	**22.03**	67.66	0.0000	0.8654	0.0000	0.8658
	2000	484.51	**50.80**	150.57	0.0000	0.8658	0.0000	0.8658
	5000	2776.46	**146.15**	434.76	0.0000	0.8658	0.0000	0.8658
	10000	10190.99	**332.34**	990.50	0.0000	0.8658	0.0000	0.8658
	15000	22372.77	**546.68**	1842.28	0.0000	0.8658	0.0000	0.8658
	20000	39157.61	**788.84**	2550.49	0.0000	0.8658	0.0000	0.8658
4	500	30.82	**15.15**	33.90	0.0000	0.8220	0.0000	0.8658
	1000	136.47	**35.20**	74.88	0.0000	0.8658	0.0000	0.8658
	2000	568.14	**84.39**	173.33	0.0000	0.8658	0.0000	0.8658
	5000	3209.42	**255.38**	504.40	0.0000	0.8658	0.0000	0.8658
	10000	11958.25	**603.19**	1141.58	0.0000	0.8658	0.0000	0.8624
	15000	26301.38	**1028.92**	2136.11	0.0000	0.8658	0.0000	0.8641
	20000	46700.79	**1537.45**	2975.18	0.0000	0.8658	0.0000	0.8658
5	500	35.50	**23.55**	40.17	0.0000	0.8322	0.0000	0.8658
	1000	156.48	**56.48**	87.91	0.0000	0.8658	0.0000	0.8657
	2000	666.15	**140.33**	200.98	0.0000	0.8658	0.0000	0.8641
	5000	3718.41	**449.71**	605.54	0.0000	0.8658	0.0000	0.8568
	10000	13815.72	**1096.25**	1367.65	0.0000	0.8658	0.0000	0.8641
	15000	30474.20	**1904.76**	2448.46	0.0000	0.8658	0.0000	0.8606
	20000	54599.66	**2879.91**	3399.76	0.0000	0.8658	0.0000	0.8639
10	500	**60.04**	114.19	69.03	0.0000	0.8602	0.0000	0.8658
	1000	263.29	366.03	**165.59**	0.0000	0.8658	0.0000	0.8658
	2000	1063.11	982.16	**380.31**	0.0000	0.8658	0.0000	0.8658
	5000	6116.69	3216.35	**1129.49**	0.0000	0.8658	0.0000	0.8658
	10000	23420.62	8056.34	**2654.56**	0.0000	0.8658	0.0000	0.8658
	15000	52877.05	14336.57	**4923.99**	0.0000	0.8658	0.0000	0.8658
	20000	95485.52	21384.65	**6948.82**	0.0000	0.8658	0.0000	0.8658
15	500	**81.78**	146.45	97.22	0.0000	0.8658	0.0000	0.8645
	1000	350.05	629.85	**236.66**	0.0000	0.8608	0.0000	0.8658
	2000	1374.82	2266.66	**583.09**	0.0000	0.8658	0.0000	0.8658
	5000	8049.52	8243.52	**1796.36**	0.0000	0.8658	0.0000	0.8658
	10000	31545.61	21661.91	**4229.87**	0.0000	0.8658	0.0000	0.8658
	15000	72274.64	38793.33	**8074.45**	0.0000	0.8658	0.0000	0.8658
	20000	137040.00	58535.86	**11391.41**	0.0000	0.8658	0.0000	0.8658

and Whitney, 1947)) to analyze the significance of differences between the SWFC-ART and FSCS-ART data, and between the SWFC-ART and LimBal-KDFC data. For two random samples, the Wilcoxon rank-sum test returns a z-statistic which is then converted into a p-value (probability value). For a 95% confidence interval (or 5% significance level), a z-statistic $\geq$ 1.96, or a p-value $\leq$ 0.01, means that there is sufficient evidence to reject the *null hypothesis* (H_0) (Ackah-Arthur et al., 2019). The H_0 states that there is no significant difference between the observed values of the two samples (Wilcoxon, 1945). The *effect size* (Sullivan and Feinn, 2012) is used to calculate the impact of the results of the experiment on an evaluation metric. The effect size for the Wilcoxon rank-sum test was calculated as (Pallant, 2016):

$$r = \frac{|z|}{\sqrt{n_1 + n_2}} \tag{4}$$

where z is the z-statistic returned by the rank-sum test and n_1 and n_2 are the sample sizes. Cohen (Cohen, 1988) identified effect sizes as *large* for $r = 0.5$; *medium* for $r = 0.3$; and *low* for $r = 0.1$.

4.5. Experimental environment

Java 1.8.0_221 was the programming language used to develop and run the simulations and experiments.[4] Two machines were used to conduct the study, both of which ran under the Microsoft Windows 10 Pro 64-bit operating system.

- Machine 1: Acer Aspire V3-572G, Intel® Core™ i5-5200U CPU @ 2.2 GHz, 2 Cores, 4 Logical Processors, 12 GB RAM.
- Machine 2: Dell OptiPlex 7050, Intel® Core™ i7-7700 CPU @ 3.60 GHz, 4 Cores, 8 Logical Processors, 16 GB RAM.

The simulations and most of the studies with the subject programs were conducted on Machine 1. However, due to the huge size and prohibitive time required, *java.util.Arrays* and *period* were tested on Machine 2. The experimental parameters were set as described in Section 3.3.

[4] We have released the SWFC-ART source code, and made it available online: https://github.com/ashfaq92/swfc-art.

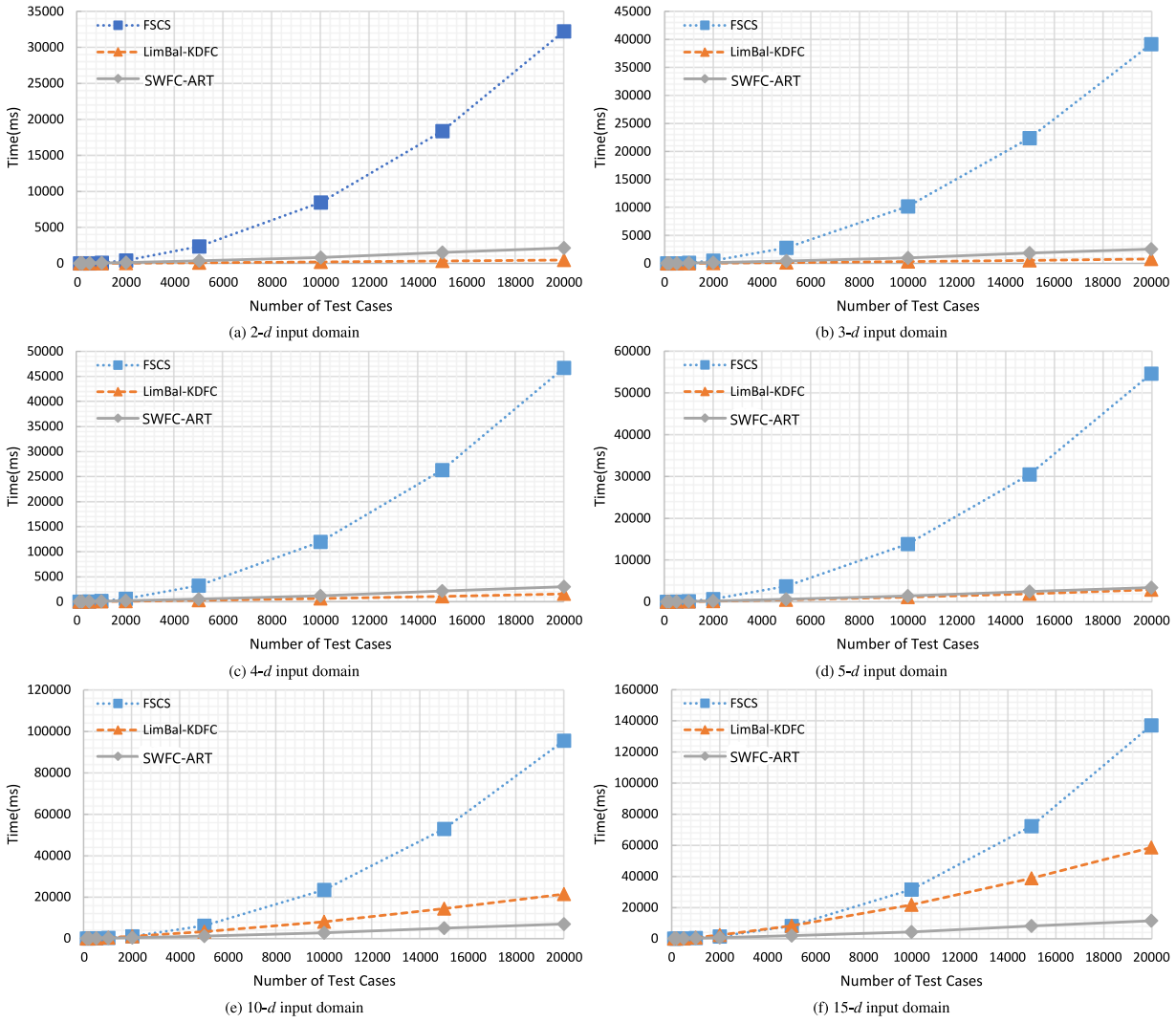

Fig. 6. Test case generation times for FSCS-ART, LimBal-KDFC and SWFC-ART, for different input domain dimensions.

5. Experimental results

5.1. Simulations

5.1.1. Computational efficiency

Table 2 presents the efficiency results, with d and n denoting the dimensionality of the SUT input domain and the number of test cases generated, respectively. All p-values are less than 0.05, and all effect sizes are greater than 0.5, which means that the SWFC-ART test case generation times are significantly different from those of FSCS-ART and LimbBal-KDFC, with large effect sizes. These results are next discussed from the perspective of the double-tier efficiency problem:

Efficiency Tier-1 (T_G trends when changing "n" within a fixed "d"): The overall trend for generating n test cases, as shown in Fig. 6, is that the FSCS-ART time complexity grows in quadratic order while the complexities of SWFC-ART and LimBal-KDFC grow in log-linear order, $O(n \cdot \log n)$.

For a 2-dimensional (2-d) input domain (Fig. 6(a)), FSCS-ART outperforms SWFC-ART when only a few test cases are generated — when $n \leq 500$, for example, FSCS-ART performs 7% faster than SWFC-ART. However, the advantage of SWFC-ART starts becoming apparent when larger numbers of tests are generated — when $n = 20{,}000$, for example, SWFC-ART shows a 93% improvement over FCSC-ART. Although both LimBal-KDFC and SWFC-ART have log-linear growth, LimBal-KDFC has a flatter slope, making it the most efficient method in 2-d input domains.

A similar trend to the 2-d observations continues until $d = 5$ (Fig. 6(d)), where, when $n > 500$, SWFC-ART is considerably more efficient than FSCS-ART, and LimBal-KDFC remains the most efficient method. A close look at Figs. 6(a), 6(b), 6(c) and 6(d), however, shows that the T_G of LimBal-KDFC starts to rise, and the gap between SWFC-ART and LimBal-KDFC decreases with the increasing dimensionality of the input domain — when generating 20,000 test cases, for example, the performance difference between SWFC-ART and LimBal-KDFC decreases from 78% to 15%, from 2-d to 5-d input domains. When $d = 5$, LimBal-KDFC and SWFC-ART appear to have the same performance — when $d > 5$ (Figs. 6(e) and 6(f)), LimBal-KDFC, which had the best efficiency in low dimensions, is now outperformed by SWFC-ART.

Interestingly, when $d = 10$ and $n \leq 1000$, FSCS-ART (with quadratic complexity) performs better than LimBal-KDFC. However, SWFC-ART maintains its scalability and efficiency, and outperforms the other methods — at $d = 10$, for example, when generating 20,000 test cases, SWFC-ART performs 92% and 67% faster than FSCS-ART and LimBal-KDFC, respectively. When $d = 15$, the LimBal-KDFC performance is worse than FSCS-ART until $n = 5000$, but the SWFC-ART performance remains consistent — when generating 20,000 test cases, SWFC-ART performs 91% and 80% faster than FSCS-ART and LimBal-KDFC, respectively.

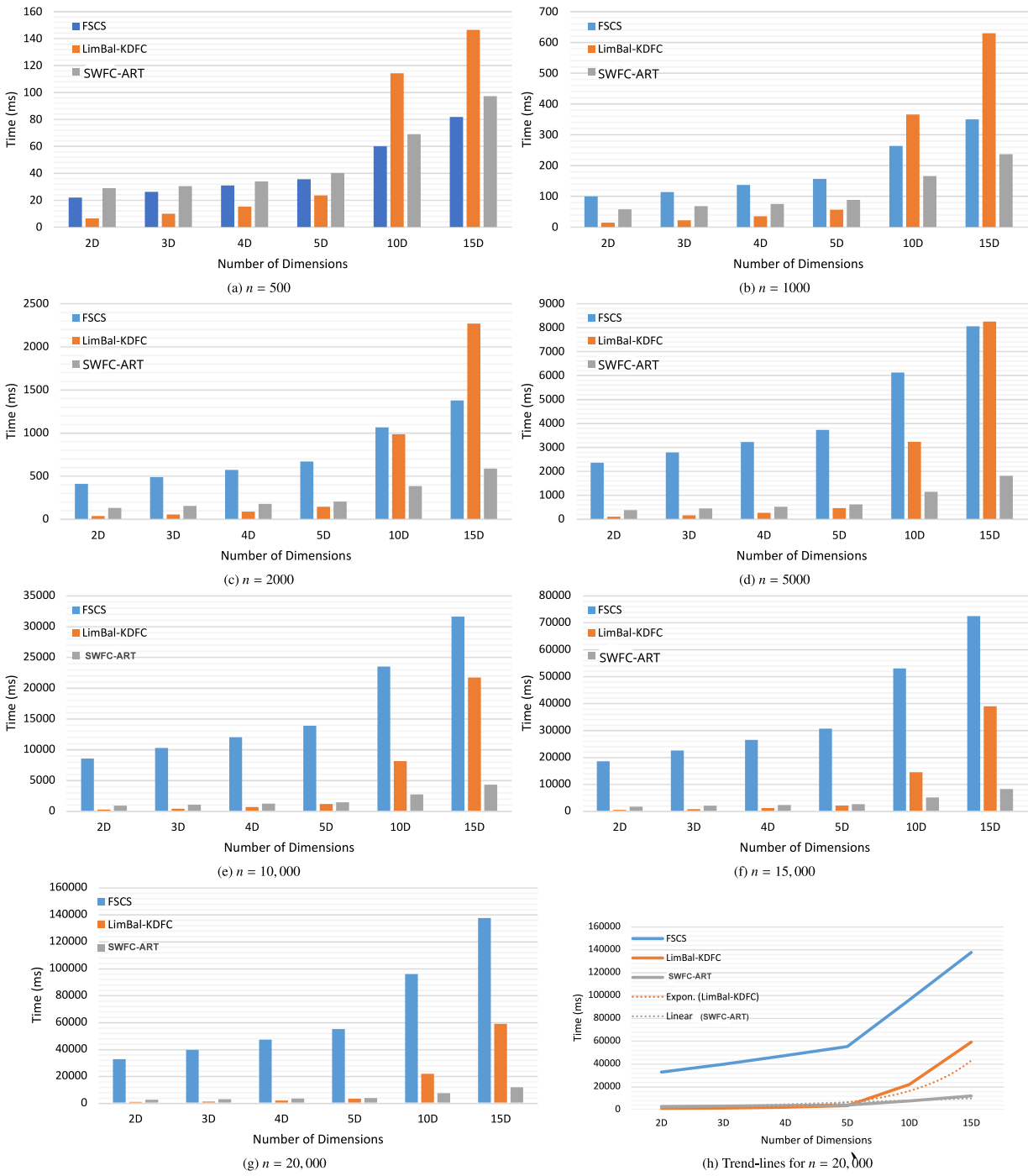

Fig. 7. Time taken to generate a fixed number of test cases, in various dimensions.

In summary, the performance improvement of SWFC-ART over FSCS-ART is always consistent (greater than 90% for n=20,000, irrespective of the input domain dimensionality). LimBal-KDFC, however, shows inconsistency, including that the rate of improvement of SWFC-ART over LimBal-KDFC when $d > 5$ is greater than that of LimBal-KDFC over SWFC-ART when $d \leq 5$.

Efficiency Tier-2 (T_G trends when changing "d" for fixed "n"): Fig. 7 shows the time taken to generate n test cases in 2-, 3-, 4-, 5-, 10- and 15-dimensional input domains. SWFC-ART is the worst performer when generating 500 test cases in a 2-d input domain (Fig. 7(a)). As the dimensionality increases, the LimBal-KDFC's T_G increases at a higher rate than the other two methods. When generating 20,000 test cases (Fig. 7(g)), although FSCS-ART has the highest T_G, its rate of increase, as the dimensionality increases, is lower than that of LimBal-KDFC. Overall, although SWFC-ART has poor relative performance when generating a small number of test cases in low dimensional input spaces, it gradually becomes faster and more consistent when the number of test cases and input domain dimensionality increase.

In general, the time taken to generate n test cases while moving from $d = 2$ to $d = 4$, by FSCS-ART, LimBal-KDFC, and SWFC-ART, increases 1.37, 2.5 and 1.32 times, respectively; when moving from $d = 5$ to $d = 10$, they increase 1.68, 5.68 and 1.83 times, respectively. The time taken by LimBal-KDFC to generate n test cases rises at the fastest rate of all three methods. Fig. 7(h) summarizes the T_G trends for $n = 20,000$ for the

Table 3
F-ratios, Wilcoxon Rank-Sum Tests, and Effect Size Analyses for FSCS-ART, LimBal-KDFC and SWFC-ART for *Block* Failure Patterns.

d	θ	F-ratio (%)			FSCS-ART vs SWFC-ART		LimBal-KDFC vs SWFC-ART	
		FSCS-ART	LimBal-KDFC	SWFC-ART	p-value	effect size	p-value	effect size
2	0.0100	69.68	**67.34**	68.66	0.2724	0.0078	0.0236	0.0160
	0.0050	**66.08**	66.15	66.86	0.3867	0.0061	0.5118	0.0046
	0.0020	**64.08**	65.21	64.51	0.7632	0.0021	0.2016	0.0090
	0.0010	**63.80**	63.85	64.29	0.5442	0.0043	0.4965	0.0048
	0.0005	64.07	64.75	**63.21**	0.2354	0.0118	0.0444	0.0200
	0.0002	64.16	**62.79**	63.10	0.1556	0.0141	0.8199	0.0022
	0.0001	**61.53**	62.55	62.98	0.2057	0.0126	0.8351	0.0020
3	0.0100	85.75	**83.65**	85.54	0.6644	0.0031	0.0517	0.0138
	0.0050	80.99	**80.18**	81.82	0.5225	0.0045	0.0816	0.0123
	0.0020	**76.97**	77.65	77.49	0.7567	0.0022	0.3305	0.0069
	0.0010	75.46	**74.79**	75.88	0.2838	0.0076	0.0858	0.0121
	0.0005	73.73	**73.53**	73.88	0.4099	0.0058	0.8224	0.0016
	0.0002	72.72	**71.45**	72.27	0.9138	0.0008	0.2490	0.0082
	0.0001	**71.36**	73.24	71.88	0.8894	0.0014	0.2246	0.0121
4	0.0100	106.86	**105.25**	106.30	0.2040	0.0090	0.8604	0.0012
	0.0050	100.79	**98.86**	100.37	0.7294	0.0024	0.2443	0.0082
	0.0020	94.19	**91.87**	93.66	0.3998	0.0060	0.1958	0.0091
	0.0010	90.99	**88.82**	90.13	0.8465	0.0014	0.1374	0.0105
	0.0005	86.77	**86.55**	87.78	0.6803	0.0029	0.4614	0.0052
	0.0002	84.01	**82.83**	84.11	0.1276	0.0196	0.6135	0.0051
	0.0001	**80.34**	82.00	83.44	0.0482	0.0255	0.1773	0.0135
5	0.0100	133.93	**127.96**	129.21	0.0261	0.0157	0.8760	0.0011
	0.0050	125.75	**118.50**	122.12	0.0550	0.0136	0.0916	0.0119
	0.0020	116.38	**109.38**	111.52	0.0038	0.0205	0.0816	0.0123
	0.0010	107.97	**105.29**	105.82	0.1404	0.0104	0.6252	0.0035
	0.0005	105.38	**98.58**	102.45	0.0187	0.0166	0.0124	0.0177
	0.0002	100.55	**96.54**	97.79	0.0146	0.0173	0.5037	0.0047
	0.0001	96.49	**92.39**	94.21	0.0886	0.0120	0.2567	0.0080
10	0.0100	405.86	392.27	**350.56**	0.0000	0.0738	0.0000	0.0582
	0.0050	365.57	339.91	**305.96**	0.0000	0.0841	0.0000	0.0659
	0.0020	313.77	268.07	**259.12**	0.0000	0.0989	0.0011	0.0324
	0.0010	290.71	236.69	**227.99**	**0.0000**	**0.1105**	0.0000	0.0430
	0.0005	266.85	**203.16**	213.68	0.0000	0.0980	0.1833	0.0133
	0.0002	242.48	**180.95**	195.57	0.0000	0.0933	0.0032	0.0295
	0.0001	220.75	**169.04**	180.67	**0.0030**	**0.1381**	0.0064	0.0273

three methods: When $d \leq 5$, all methods show monotonous growth, but when $d > 5$, they all appear to encounter the *curse of dimensionality* (Domingos, 2012; Bellman, 1957, 1954), with T_G values increasing sharply for FSCS-ART and LimBal-KDFC, but SWFC-ART remaining consistent.

5.1.2. Failure-detection effectiveness

The simulation results for the block failure pattern are reported in Table 3. For the 2-d input domain, *all* p-values are much greater than 0.05 and all effect sizes are less than 0.1, meaning that no significant difference exists between the F-ratios of FSCS-ART, LimBal-KDFC, and SWFC-ART. A similar trend can be seen for the 3- and 4-d input domains. The trend is also present in the 5-d input domain, except when $\theta = 0.002$, for which the p-value for FSCS-ART and SWFC-ART is 0.0038 — although the effect size is still less than 0.5 (0.0205) — which means that there is insufficient evidence to conclude whether or not the F-ratios are different. In the 10-d input domain, the p-values are less than 0.05, meaning that the F-ratios are significantly different, with SWFC-ART outperforming FSCS-ART. However, the effect sizes are still far less than 0.05, which means that even if the samples are different, there is still only a negligible effect on the F-ratios. In summary, there was insufficient evidence to reject H_0 and thus we conclude that the F-ratio results for all three methods are similar for the block failure pattern.

Table 4 shows the strip failure pattern simulation results. For all dimensions, the p-values are greater than 0.05 and the effect sizes are less than 0.1, which again means that there is insufficient evidence to reject the null hypothesis.

The point failure pattern simulation results, shown in Table 5, present similar trends to those seen in the block pattern results. In low dimensional input domains ($d \leq 5$), the p-values and effect sizes show that the F-ratios of all three methods are similar. In the 10-d input domain, the p-values show a significant difference, especially between SWFC-ART and FSCS-ART. The mean SWFC-ART F-ratios are better than those of FSCS-ART, and significantly better than LimBal-KDFC when θ=0.01, 0.005 and 0.001; LimBal-KDFC outperforms SWFC-ART when θ=0.0002 and 0.0001. However, the effect size values are not large enough to allow strong conclusions to be drawn.

In summary, we can conclude that the F-ratios of all three methods are similar, for all the failure rates, failure patterns, and input dimensions under study. Furthermore, the ANNS strategies (LimBal-KDFC and SWFC-ART) performed better than the exact NNS of FSCS-ART in high dimensional input spaces. The ANNS strategies employed by LimBal-KDFC and SWFC-ART were also significantly different from each other in high dimensions, while showing similar failure-detection effectiveness.

5.1.3. Test case distribution

Table 6 shows the discrepancy values for 100, 1000 and 10,000 test cases, for all three methods. Three important trends can be observed:

1. When generating a specific number of test cases in a particular dimension, there is no significant difference in the discrepancy values for all three methods. When there is a difference, the ANNS methods (SWFC-ART and LimBal-KDFC) are usually better than the exact NNS method (FSCS-ART), with FSCS-ART having better discrepancy values only three out of 63 times. Hence, it can be concluded that the test case distribution of methods employing ANNS is at least equal to the exact NNS methods.

Table 4
F-ratios, Wilcoxon Rank-Sum Tests, and Effect Size Analyses for FSCS-ART, LimBal-KDFC and SWFC-ART for *Strip* Failure Patterns.

d	θ	F-ratio (%)			FSCS-ART vs SWFC-ART		LimBal-KDFC vs SWFC-ART	
		FSCS-ART	LimBal-KDFC	SWFC-ART	p-value	effect size	p-value	effect size
2	0.0100	93.29	91.90	**91.59**	0.1827	0.0094	0.6709	0.0030
	0.0050	94.48	**94.25**	94.37	0.4042	0.0059	0.7006	0.0027
	0.0020	97.85	**96.14**	97.76	0.6088	0.0036	0.0933	0.0119
	0.0010	98.25	99.85	**96.85**	0.4100	0.0058	0.0296	0.0154
	0.0005	98.86	98.34	**95.36**	0.1472	0.0144	0.5097	0.0065
	0.0002	97.62	**95.91**	101.04	0.4311	0.0078	0.2896	0.0105
	0.0001	100.38	101.53	**98.21**	0.4917	0.0068	0.1760	0.0135
3	0.0100	97.01	97.84	**96.53**	0.5924	0.0038	0.7784	0.0020
	0.0050	**98.09**	98.65	98.96	0.6399	0.0033	0.7847	0.0019
	0.0020	99.63	99.64	**98.83**	0.9175	0.0007	0.8003	0.0018
	0.0010	99.94	**98.92**	99.45	0.5357	0.0044	0.5505	0.0042
	0.0005	**98.99**	99.93	99.92	0.5154	0.0046	0.4547	0.0053
	0.0002	99.23	100.86	**98.82**	0.6649	0.0031	0.0176	0.0168
	0.0001	101.24	101.31	**101.16**	0.8457	0.0019	0.5195	0.0064
4	0.0100	99.74	**98.96**	101.09	0.1633	0.0099	0.0233	0.0160
	0.0050	99.62	**99.23**	99.86	0.9521	0.0004	0.2178	0.0087
	0.0020	**98.52**	100.58	100.78	0.0752	0.0126	0.3637	0.0064
	0.0010	**98.40**	99.93	99.99	0.7849	0.0019	0.8062	0.0017
	0.0005	**99.87**	100.22	99.91	0.4659	0.0052	0.6741	0.0030
	0.0002	**98.68**	101.92	98.98	0.4159	0.0105	0.0936	0.0168
	0.0001	101.15	99.39	**97.10**	0.1709	0.0176	0.0658	0.0184
5	0.0100	100.48	**98.75**	101.88	0.2224	0.0086	0.0512	0.0138
	0.0050	**99.68**	99.87	99.77	0.8492	0.0013	0.4507	0.0053
	0.0020	99.41	**98.36**	100.22	0.4515	0.0053	0.0414	0.0144
	0.0010	**98.66**	100.89	100.55	0.3531	0.0066	0.9837	0.0001
	0.0005	100.96	99.71	**98.77**	0.2659	0.0079	0.7164	0.0026
	0.0002	100.02	99.17	**99.01**	0.3432	0.0067	0.9036	0.0009
	0.0001	99.26	**98.57**	100.08	0.8683	0.0017	0.3254	0.0098
10	0.0100	**99.26**	101.28	103.13	0.2442	0.0116	0.0966	0.0166
	0.0050	102.04	**101.12**	101.21	0.9038	0.0012	0.1216	0.0154
	0.0020	99.38	**97.64**	100.69	0.6301	0.0048	0.3151	0.0100
	0.0010	**94.90**	101.24	96.10	0.2681	0.0143	0.0621	0.0187
	0.0005	100.31	101.25	**100.19**	0.7399	0.0033	0.3629	0.0090
	0.0002	**100.52**	100.60	101.03	0.2239	0.0157	0.9277	0.0009
	0.0001	101.70	99.68	**99.24**	0.6137	0.0331	0.6475	0.0046

2. All test case generation strategies appear to display degradation in distribution with increasing dimensionality of the input domain: The SWFC-ART discrepancy values, for example, increase from 0.0562 to 0.2759 from 1-*d* to 15-*d*, for 100 test cases. However, SWFC-ART appears least affected by the *curse of dimensionality* (Domingos, 2012; Bellman, 1957, 1954).
   To generate 100 test cases, SWFC-ART shows better discrepancy values in 1, 4, 10, and 15 dimensions; for 1000 test cases, SWFC-ART has better discrepancy in 3, 4, 5, and 10 dimensions; and for 10,000 test cases, SWFC-ART has better discrepancy in 3, 4, 10, and 15 dimensions.
3. All methods showed better discrepancy values as the number of generated test cases increased.

5.2. Experiments with real-life programs

5.2.1. Computational efficiency

The time taken to test a program is divided into two parts: the test case generation time (T_G) and the execution time (T_E). When testing the real-life programs, test cases were incrementally generated and executed until a failure was revealed—output of the original program was different from its fault-seeded version.

As can be seen in Table 7, the T_E values for all the programs under study were far less than the T_G values, with the T_E results for all methods being comparable. T_G is, therefore, the main time cost for FSCS-ART, and any reduction in T_G should have a positive impact on FSCS-ART efficiency, especially when $T_E < T_G$.

SWFC-ART has significantly lower T_G results than FSCS-ART for all the programs except *tanh*, with results for the programs *airy*, *period*, *pntTrianglePos*, and *twoLinesPos* being particularly dramatic (reductions of about 90%). SWFC-ART outperforms FSCS-ART by 80%–90% for the programs *asinh*, *binomial*, *matrixProcessor*, *java.util.Arrays*, *line*, *tcas*, *select*, and *cel*; and 70%–80% for *complex*, *triangle*, *CalDay*, *pntLinePos*, *plgndr*, and *golden*. Overall, SWFC-ART reduced the FSCS-ART T_G by more than 50% for *all* the programs under study except *bessj*, *bessj0* and *tanh* — where the performance improvement was 29%, 8% and -34%, respectively.

SWFC-ART performs a little worse than LimBal-KDFC in (programs with) low dimensions, with five or less input parameters, but has better performance in the high dimensional programs ($d > 5$) — *matrixProcessor, java.util.Arrays*, *select*, *calGCD*, *nearestDistance*, *line*, *twoLinesPos*, and *PntTrianglePos* where SWFC-ART outperforms LimBal-KDFC by 73%, 68%, 68%, 59%, 57%, 49%, 48%, and 46%, respectively. SWFC-ART can also sometimes outperform LimBal-KDFC in low dimensional programs (*airy*); and LimBal-KDFC can sometimes perform better than SWFC-ART in some high dimensional programs (*triangle*, and *tcas*). For *complex* and *pntLinePos*, both methods show similar T_G.

In summary, the results of the experimental studies with real-life programs are consistent with the simulations: Both LimBal-KDFC and SWFC-ART outperform FSCS-ART in terms of computational efficiency, and SWFC-ART remains consistent in high dimensional programs.

5.2.2. Failure-detection effectiveness

Table 8 presents the F-measure effectiveness results, showing that, for the 28 programs, FSCS-ART, LimBal-KDFC, and SWFC-ART have the best results in 8, 12, and 8 programs, respectively.

Table 5
F-ratios, Wilcoxon Rank-Sum Tests, and Effect Size Analyses for FSCS-ART, LimBal-KDFC and SWFC-ART for *Point* Failure Patterns.

d	θ	F-ratio (%)			FSCS-ART vs SWFC-ART		LimBal-KDFC vs SWFC-ART	
		FSCS-ART	LimBal-KDFC	SWFC-ART	p-value	effect size	p-value	effect size
2	0.0100	**99.18**	99.62	103.04	0.0054	0.0197	0.0280	0.0155
	0.0050	100.10	99.44	**99.19**	0.5853	0.0039	0.8042	0.0017
	0.0020	**96.31**	98.37	97.68	0.3614	0.0065	0.8473	0.0013
	0.0010	97.79	98.38	**97.75**	0.4763	0.0050	0.5360	0.0043
	0.0005	98.31	96.50	**96.46**	0.6352	0.0047	0.7385	0.0033
	0.0002	94.88	95.60	**94.41**	0.7026	0.0038	0.4861	0.0069
	0.0001	**95.67**	97.01	96.10	0.9466	0.0006	0.4030	0.0083
3	0.0100	112.00	112.25	**111.65**	0.6711	0.0030	0.5859	0.0039
	0.0050	**106.91**	107.81	108.71	0.4195	0.0057	0.9024	0.0009
	0.0020	105.68	104.67	**104.63**	0.3997	0.0060	0.8001	0.0018
	0.0010	102.32	102.37	**102.19**	0.6468	0.0032	0.8468	0.0014
	0.0005	101.63	101.58	**100.29**	0.3700	0.0063	0.5001	0.0048
	0.0002	**99.26**	101.21	99.65	0.6180	0.0035	0.4311	0.0056
	0.0001	98.36	**97.88**	99.24	0.6133	0.0065	0.6882	0.0040
4	0.0100	129.63	**127.25**	128.92	0.5684	0.0040	0.1229	0.0109
	0.0050	125.38	124.26	**123.47**	0.8625	0.0012	0.8929	0.0010
	0.0020	117.24	**116.33**	116.41	0.5400	0.0043	0.9631	0.0003
	0.0010	114.61	**113.50**	113.83	0.8201	0.0016	0.5857	0.0039
	0.0005	**107.00**	108.62	110.34	0.4013	0.0108	0.6633	0.0044
	0.0002	107.79	105.65	**105.60**	0.4562	0.0096	0.6999	0.0039
	0.0001	**105.59**	106.37	106.20	0.6547	0.0057	0.7695	0.0029
5	0.0100	153.57	150.38	**149.39**	0.0383	0.0146	0.2678	0.0078
	0.0050	145.70	141.07	**140.73**	0.0062	0.0193	0.8661	0.0012
	0.0020	134.72	**130.30**	130.71	0.0444	0.0142	0.7011	0.0027
	0.0010	129.35	126.02	**125.73**	0.1885	0.0093	0.5049	0.0047
	0.0005	124.58	**121.15**	122.53	0.0752	0.0126	0.5975	0.0037
	0.0002	119.99	**114.77**	119.38	0.2236	0.0086	0.0620	0.0132
	0.0001	115.45	**111.16**	114.53	0.6723	0.0054	0.0952	0.0167
10	0.0100	252.49	247.17	**237.79**	0.7246	0.0045	0.3984	0.0084
	0.0050	278.97	271.20	**244.12**	0.0000	0.0586	0.0000	0.0494
	0.0020	292.82	265.93	**240.21**	0.0000	0.0852	0.0000	0.0641
	0.0010	291.09	244.76	**236.16**	0.0000	0.0974	0.0038	0.0289
	0.0005	272.80	233.53	**227.46**	0.0000	0.0840	0.0441	0.0201
	0.0002	242.80	**206.08**	209.57	**0.0000**	**0.1109**	0.6922	0.0040
	0.0001	236.35	**191.35**	192.74	0.0000	0.0194	0.5468	0.0060

Table 6
Discrepancy in Various Dimensions.

Test cases	Method	Discrepancy						
		1-d	2-d	3-d	4-d	5-d	10-d	15-d
100	FSCS-ART	0.0750	0.1393	**0.2159**	0.2697	**0.3112**	0.3135	0.3108
	LimBal-KDFC	0.0628	**0.1295**	0.2250	0.2856	0.3138	0.3070	0.2886
	SWFC-ART	**0.0562**	0.1359	0.2420	**0.2652**	0.3206	**0.2942**	**0.2759**
1000	FSCS-ART	0.0181	0.0381	0.1036	0.1620	0.1899	0.2228	0.2090
	LimBal-KDFC	**0.0163**	**0.0347**	0.0961	0.1592	0.1884	0.2140	**0.2055**
	SWFC-ART	0.0168	0.0355	**0.0930**	**0.1515**	**0.1739**	**0.2010**	0.2106
10,000	FSCS-ART	**0.0084**	0.0163	0.0574	0.1091	0.1534	0.1987	0.1889
	LimBal-KDFC	0.0085	**0.0133**	0.0560	0.1098	**0.1385**	0.1883	0.1859
	SWFC-ART	0.0089	0.0151	**0.0525**	**0.1033**	0.1388	**0.1795**	**0.1623**

Comparing FSCS-ART and SWFC-ART: The Wilcoxon rank-sum tests for the FSCS-ART and SWFC-ART F-measure data have p-values greater than 0.05 for all subject programs, except *twoLinesPos*, with extremely low effect sizes — although the p-value for *twoLinesPos* is 0.0199, the effect size is much less than 0.1. Moreover, for *twoLinesPos*, the mean F-measure of SWFC-ART is 7613.96 while that for FSCS-ART is 8177.49. The F-measure results for *sncndn* are very similar for both approaches; for 13 programs (*airy, probks, gammq, binomial, golden, el2, calDay, triangle, line, nearestDistance, tcas, matrixProcessor*, and *java.util.Arrays*), FSCS-ART has better results than SWFC-ART; and for the remaining 15 programs (*bessj0, asinh, erfcc, tanh, bessj, plgndr, cel, period, complex, pntLinePos, pntTrianglePos, twoLinesPos, calGCD* and *select*), SWFC-ART has the better performance. However, the p-value and effect size analyses are such that it is not statistically clear that either method does actually perform better than the other.

Comparing LimBal-KDFC and SWFC-ART: The p-values for the comparisons of the LimBal-KDFC and SWFC-ART F-measure results are all greater than 0.05 (except for *twoLinesPos* and *java.util.Arrays*), and all effect sizes are much less than 0.1. This means that the failure-detection effectiveness of both methods is similar. For *twoLinesPos* and *java.util.Arrays*, although the p-value is significant, the effect size is not large enough for any conclusion to be drawn.

In summary, the results from the studies with real-life programs again align with those of the simulations: There was insufficient evidence to reject the null hypothesis, and thus we conclude that there are no significant differences among the observed F-measures of all the methods — the failure-finding effectiveness of all methods is similar.

Table 7
Test Case Generation Time and Execution Time for Detecting Failures in the 23 Subject Programs.

Program	d	T_G (ms)			T_E (ms)		
		FSCS-ART	LimBal-KDFC	SWFC-ART	FSCS-ART	LimBal-KDFC	SWFC-ART
bessj0	1	22.14	**4.16**	20.33	1.47	1.36	1.45
airy	1	86.78	10.01	**2.23**	2.32	1.96	2.23
asinh	1	3.8e+9	**9.5e+7**	4.6e+8	3e+6	2e+6	2e+6
erfcc	1	129.65	**12.08**	57.40	4.69	3.99	4.43
probks	1	270.41	**19.87**	90.66	78.41	76.93	78.64
tanh	1	13.59	**3.61**	18.22	1.44	1.33	1.67
bessj	2	34.61	**5.56**	24.24	3.70	3.60	3.76
gammq	2	229.95	**16.60**	71.80	5.26	4.21	4.72
Sncndn	2	87.48	**10.11**	42.29	2.86	2.28	2.63
binomial	2	4.1e+9	**1.3e+8**	4.9e+8	1.4e+7	1.4e+7	1.4e+7
plgndr	3	651.81	**31.86**	136.44	21.34	18.11	19.14
golden	3	784.13	**51.98**	160.40	25.43	23.47	24.27
cel	4	702.66	**39.14**	139.56	9.63	7.41	8.09
el2	4	157.08	**29.26**	66.21	4.48	3.91	4.28
period	4	3.9e+9	**1.3e+8**	4.5e+8	1.0e+7	1.0e+7	1.0e+7
calDay	5	496.08	**51.18**	120.65	10.55	10.24	10.68
complex	6	597.86	**150.12**	155.63	0.75	0.61	0.62
pntLinePos	6	968.20	**206.68**	209.48	0.40	0.24	0.27
triangle	6	774.64	**179.19**	191.10	0.86	0.52	0.52
line	8	5097.56	1311.34	**656.25**	0.75	0.52	0.48
pntTrianglePos	8	15699.36	2449.96	**1311.27**	6.94	4.16	3.82
twoLinesPos	8	42078.38	4656.06	**2403.85**	8.24	4.31	4.08
nearestDistance	10	668.51	552.93	**232.98**	0.66	0.61	0.56
calGCD	10	522.59	440.67	**179.26**	3.29	3.33	3.31
select	11	4508.39	2333.18	**744.19**	6.25	5.75	5.01
tcas	12	2310.45	**92.20**	348.51	1.46	0.95	0.90
matrixProcessor	12	1.5e+10	7.0e+9	**1.8e+9**	8.0e+7	8.0e+7	7.0e+7
java.util.Arrays	15	1.9e+10	7.6e+9	**2.4e+9**	2.0e+8	2.0e+8	2.0e+8

Table 8
Wilcoxon Rank-Sum Test and Effect Size Analyses of F-measure of FSCS-ART, LimBal-KDFC, and SWFC-ART for Subject Programs.

Program	d	F-measure			FSCS-ART vs SWFC-ART		LimBal-KDFC vs SWFC-ART	
		FSCS-ART	LimBal-KDFC	SWFC-ART	p-value	effect size	p-value	effect size
bessj0	1	444.46	448.53	**440.48**	0.5161	0.0065	0.2342	0.0119
airy	1	**789.54**	806.50	807.18	0.1045	0.0162	0.6903	0.0040
asinh	1	5689	5612	**5583**	0.0773	0.0177	0.5439	0.0061
erfcc	1	1037.81	**1024.31**	1032.24	0.3863	0.0087	0.8743	0.0016
probks	1	**1453.62**	1460.71	1456.30	0.9354	0.0008	0.7251	0.0035
tanh	1	313.05	311.31	**310.70**	0.5812	0.0055	0.7955	0.0026
bessj	2	454.54	**440.02**	442.08	0.1974	0.0129	0.3242	0.0099
gammq	2	1086.95	**1045.17**	1097.29	0.9064	0.0012	0.1866	0.0132
sncndn	2	631.45	631.47	**631.23**	0.4489	0.0076	0.8246	0.0022
binomial	2	**5089**	5243	5166	0.6713	0.0042	0.5927	0.0053
plgndr	3	1618.00	**1606.05**	1608.47	0.7092	0.0037	0.8469	0.0019
golden	3	**1802.60**	1808.43	1804.03	0.8994	0.0013	0.8369	0.0021
cel	4	1547.32	1571.71	**1542.93**	0.6888	0.0040	0.1664	0.0138
el2	4	**721.55**	724.94	728.06	0.8781	0.0015	0.3901	0.0086
period	4	29794	30307	**29765**	0.5266	0.0091	0.0321	0.0309
calDay	5	**1259.38**	1314.12	1280.12	0.8884	0.0014	0.0768	0.0177
complex	6	1223.95	1214.50	**1195.77**	0.2619	0.0112	0.2391	0.0118
pntLinePos	6	1503.62	**1462.88**	1477.83	0.9587	0.0005	0.3696	0.0090
triangle	6	**1350.33**	1389.29	1379.52	0.0783	0.0176	0.6655	0.0043
line	8	3370.27	**3326.32**	3385.39	0.3786	0.0088	0.3627	0.0091
pntTrianglePos	8	4713.73	**4238.11**	4657.62	0.4050	0.0107	0.0593	0.0242
twoLinesPos	8	8177.49	**7009.96**	7613.29	0.0199	0.0301	0.0298	0.0280
nearestDistance	10	**1934.32**	2015.68	2065.56	0.0716	0.0233	0.3113	0.0131
calGCD	10	1035.47	1023.50	**1003.32**	0.4538	0.0097	0.6892	0.0052
select	11	5583.22	**5174.07**	5432.77	0.7461	0.0042	0.0506	0.0252
tcas	12	1681.02	**1642.95**	1736.33	0.3609	0.0118	0.1526	0.0185
matrixProcessor	12	5003	**4978**	5152	0.2663	0.0111	0.4101	0.0082
java.util.Arrays	15	10108	**9866**	10313	0.0994	0.0164	0.0198	0.0233

5.3. Discussion

In this section, we summarize our results by giving answers to the research questions. We also discuss the asymptotic complexities of the methods.

5.3.1. RQ1

Efficiency Tier-1 (Scalability): Our studies have shown that, as the number of executed tests increases, FSCS-ART incurs considerable time overheads, but LimBal-KDFC and SWFC-ART perform

significantly better. This can also be understood through a theoretical analysis of the complexities of the algorithms: The FSCS-ART time complexity is in quadratic relation with n, but both LimBal-KDFC and SWFC-ART are in a log-linear relation. Because real-life software can typically have very low failure rates (Arcuri and Briand, 2011), it may be necessary to generate and execute many test cases before finding a first failure — LimBal-KDFC and SWFC-ART, therefore, would perform much faster than FSCS-ART in such situations.

Efficiency Tier-2 (Consistency): Our studies have also shown that all three methods take increasing amounts of time to generate a fixed number of test cases in increasing dimensional input spaces. When there were more than five SUT input parameters, the *curse of dimensionality* (Domingos, 2012; Bellman, 1957, 1954) started to impact on the performance of LimBal-KDFC and FSCS-ART, but SWFC-ART retained consistency (LimBal-KDFC was the least consistent). Again, this can be understood from the theoretical complexity analysis, where LimBal-KDFC has quadratic, while FSCS-ART and SWFC-ART have a linear dimensional dependence on their time complexities. As real-life programs typically have high-dimensional input domains (many program input parameters) (Lin et al., 2009), SWFC-ART would perform much faster than the other two methods in such situations.

Answer to RQ1: *FSCS-ART has a quadratic time complexity relation with the number of executed tests (n), and LimBal-KDFC has a quadratic time complexity relation with the dimensionality (d) of the SUT. Because real-life programs may often have low failure rates and high dimensional input spaces, neither FSCS-ART nor LimBal-KDFC solve the double-tier efficiency problem. In contrast, SWFC-ART's consistency and scalability, regardless of n and d, make it the preferred method.*

5.3.2. RQ2

The results show that the failure-detection effectiveness of SWFC-ART is similar to that of FSCS-ART and LimBal-KDFC. No significant deterioration in the failure-detection effectiveness was observed — the null hypothesis (H_0) could not be rejected.

We also found that the introduction of ANNS strategies delivered similar failure-detection effectiveness to NNS, while significantly reducing the computational overhead. Although LimBal-KDFC implements the ANNS through a limited back-tracking strategy and SWFC-ART maintains a Delaunay graph at its bottom layer, both ANNS strategies have similar failure-detection effectiveness.

Answer to RQ2: *SWFC-ART employs a unique ANNS strategy and its failure-detection effectiveness is comparable to that of the state-of-the-art method, KDFC-ART.*

5.3.3. RQ3

The discrepancy values for SWFC-ART in the experiments were usually better than those for FSCS-ART and LimBal-KDFC, with no observed significant deterioration in even-spreading. We can therefore conclude that, although SWFC-ART incorporates an ANNS, its test case distribution remains comparable to that of the state-of-the-art method, KDFC-ART.

Answer to RQ3: *SWFC-ART distributes test cases as evenly as the state-of-the-art method, KDFC-ART, even though it employs an ANNS strategy.*

6. Threats to validity

This section discusses some potential limitations of our work and validity threats to our conclusions.

6.1. Construct validity

Construct validity refers to how well the evaluation measures support the investigation of the research questions (Wohlin et al., 2012). We used T_G and T_E to evaluate the efficiency of the methods (RQ1). Although *F-time* (the sum of the generation, execution, and evaluation times for a test case) (Chen and Merkel, 2006a) has also been used in some ART studies, reporting the component times separately appears to be the preferred option when evaluating FSCS-ART computational overhead reduction strategies (Huang et al., 2015; Mao et al., 2017, 2019) — this removes ambiguity in scenarios where a method may have less T_G but more T_E.

We measured the failure-detection effectiveness (RQ2) using the F-measure (or F-ratio). Alternative metrics do exist, including the *P-measure* (the probability of detecting at least one failure) and *E-measure* (the expected number of failures detected) (Chen et al., 2005, 2006b; Ackah-Arthur et al., 2019). However, both P-measure and E-measure assume that the size of the test set to be known in advance. The research in this paper involves the incremental generation of test cases by ART, and so the F-measure (or F-ratio) is the preferred measure for such comparisons. Nevertheless, we look forward to exploring alternative evaluation metrics in our future work.

In addition to discrepancy, the dispersion and edge-to-center ratio (Chen et al., 2007b; Branicky et al., 2001) have also been used to measure test case distribution (RQ3). However, the edge-to-center ratio metric is rarely used, and low discrepancy corresponds to lower dispersion (and vice versa). Given its simplicity and ease of implementation, we are confident in our choice of discrepancy as the test case distribution metric in our studies.

6.2. External validity

External validity refers to how generalizable the experimental results are for other scenarios. We used FSCS-ART and LimBal-KDFC as the baseline comparison methods — FSCS-ART was selected due to its failure-detection effectiveness, and LimBal-KDFC was selected as the state-of-the-art enhancement of FSCS-ART for both low and high dimensional programs. Although other overhead reduction strategies exist for FSCS-ART that claim to have "linear-order" time complexity, most of them have dimension-related exponential time complexity (as discussed in Section 7). Our efficiency comparisons were performed in up to 15-dimensional input spaces, with up to 20,000 test cases generated in each. We also used 28 real-life programs of various sizes and dimensions, more than most other comparable studies.

The study reported on in this paper only examined numeric programs, and only used the Euclidean distance to measure the similarity between test cases. SWFC-ART supports a wide variety of distance metrics, and so we look forward to exploring its performance with non-numeric programs and other distance metrics in our future work. Furthermore, because the scope of this study was to increase the efficiency of the FSCS-ART method by employing a graph-based approach, the effectiveness could be increased by integrating SWFC-ART with existing methods that aim to enhance the effectiveness of FSCS-ART (Li et al., 2019; Schneckenburger and Schweiggert, 2008; Kuo et al., 2008, 2007a; Chen et al., 2007c; Kuo et al., 2007b; Mayer, 2006a). In particular, given their demonstrated potential for increasing the failure-detection effectiveness without impacting the efficiency, we look forward to experimenting using the Manhattan distance (Huang et al., 2020) metric and Inverted FSCS-ART (Kuo et al., 2007a) in our future work.

6.3. Internal validity

Internal validity refers to confidence in the findings of the study. We have double-checked and cross-validated the implemented algorithms to ensure that there is no mistake in the experimental setup. The methods were rigorously evaluated under different settings — dimensions, failure patterns, number of generated test cases, real-life programs — and statistical tests were performed on the results. The number of trials (sample sizes) in the simulations and experiments were significant enough to achieve the desired confidence level. Overall, we have confidence in the correctness of our evaluation setup

7. Related work

This section highlights some related work in ART computational overhead reduction. Each method discussed is described according to its approach, time complexity, failure-detection effectiveness, and the effect of dimensionality on the method's efficiency.

7.1. FSCS-ART overhead reduction methods

7.1.1. C.G. FSCS-ART

One of the earliest attempts to reduce FSCS-ART computational overheads involved a *Center-of-Gravity* (C.G.) constraint (Chan et al., 2004). This method reduces FSCS-ART distance computations by selecting a test case from the candidate set such that the resulting C.G. is as close as possible to the input domain's C.G. Although this method was reported to reduce time overheads by 74% in a small experimental study with a real-life program, it still faces quadratic time complexity. Moreover, the method is unable to maintain the failure-finding effectiveness of the original FSCS-ART, with drops of up to 11% reported in its F-measure values.

7.1.2. D-FSCS-ART

Descending distance FSCS-ART (D-FSCS-ART) (Jixin Geng and Jiongmin Zhang, 2010) aimed to reduce the number of distance computations of FSCS-ART by sorting the elements of E according to their x-coordinates, and recording *Lastmin* (the last maximum value of the shortest distance). For each new candidate test case, only the x-distance (the difference in x-coordinates) is calculated. To calculate the maximum value of the shortest distances, E is split in half and the direction of distance calculation is determined by the x-distance. In this way, the number of distance calculations is reduced — in 4-d when $\theta = 0.001$, for example, D-FSCS-ART achieved an 18% reduction in distance calculations. However, the efficiency improvement was not discussed in terms of F-time or T_G (Section 4.2), and it was not clear whether or not the reduction in distance calculations would impact the overall efficiency. Furthermore, it was also not clear whether or not the reduction in distance calculations preserved the failure-detection effectiveness of the original FSCS-ART.

7.1.3. FSCS-ART by temporal forgetting

FSCS-ART by temporal forgetting has three types: complete restart, random forgetting, and consecutive retention (Chan et al., 2006). Complete restart simply resets the algorithm after a certain number of test case executions, determined by k, the "memory parameter". Random forgetting maintains a fixed-sized (k) executed test case set by randomly filtering out n-k executed tests. Consecutive retention limits distance computations to only the k most-recently-executed test cases. While temporal forgetting may have "linear-order" complexity regardless of the SUT dimensionality, it can suffer from severe degradation in failure-detection effectiveness. As reported, larger values for the memory parameter provide better effectiveness, but lower efficiency. Selection of an appropriate value for the memory parameter, therefore, remains challenging.

7.1.4. FSCS-ART by distance-aware forgetting

The original forgetting strategy may not make much use of information about forgotten test cases, which led to the development of Distance-aware Forgetting FSCS-ART (DF-FSCS-ART) (Mao et al., 2017), which uses executed test case spatial information. DF-FSCS-ART uses grid partitioning, with test cases located within the neighboring partition of a cell considered to be "in-sight", and those outside the partition considered "out-of-sight", and thus forgotten.

DF-FSCS-ART claims to preserve the FSCS-ART failure-detection ability, but it has a complexity of $O(\tau \cdot 3^d \cdot k \cdot n)$ — where τ is a preset constant for the dynamic partitioning; d is the input domain dimensionality; k is the the candidate set size; and n is the executed set size. Although DF-FSCS-ART can generate test cases in linear-order, it still has exponential complexity in terms of dimensionality. As we saw in our experimental studies (Section 4), even though LimBal-KDFC had a quadratic complexity relation with the SUT dimension (in the worst-case), the performance was severely impacted by the dimensionality. As most real-life software can have high dimensionality, methods that have exponential complexity with respect to the SUT dimensionality may not be practical.

7.1.5. KDFC-ART

There are three variants of the recently proposed KD-tree approach to FSCS-ART (Mao et al., 2019): Naive-KDFC; SemiBal-KDFC; and LimBal-KDFC. LimBal-KDFC is an efficient ART algorithm that attempts to resolve the dimensionality-related FSCS-ART overhead problem while maintaining comparable failure-detection effectiveness. Although LimBal-KDFC has a log-linear time complexity with respect to the number of generated test cases — which is a little worse than linear-order algorithms — it only has an $O(d^2)$ relation with the SUT dimensionality in its worst-case, which is far better than the $O(3^d)$ for DF-FSCS-ART (Section 7.1.4). LimBal-KDFC also supports incremental generation of test cases, an advantage over RBCVT-Fast (Section 7.2.1). Both LimBal-KDFC and SWFC-ART use sophisticated data structures to store the previously executed, non-failure-causing test cases, but the actual structures are very different: While LimBal-KDFC uses a KD-tree, which is efficient and straightforward, but delivers inconsistent performance in high dimensional input spaces (Lee and Wong, 1977), SWFC-ART uses a graph-based data structure that is explicitly designed for high dimensional input domains. As discussed in Section 5, LimBal-KDFC appeared to become inconsistent when the input domain dimensionality went above five, but SWFC-ART continued to perform consistently regardless of the number of dimensions.

7.2. Other efficient ART methods

7.2.1. RBCVT-Fast

RBCVT-Fast (Shahbazi et al., 2013) combines a search-based algorithm with Voronoi region centroids to reduce ART overheads. Although the method has linear-order time complexity and comparable failure-detection effectiveness, the size of the executed test set ($|E|$) must be specified in advance, which is usually impossible (because $|E|$ depends on the SUT failure rate, which is unknown before testing). Furthermore, RBCVT-Fast faces a potential risk for high dimensional programs: Because it makes use of a parameter (α) to control the probability of test cases

generated within the "random border" — an imaginary border outside the real input domain borders — if an appropriate value for the parameter is not chosen, the effectiveness may suffer (Mao et al., 2019, 2017).

7.2.2. ART by partitioning

Although ART by Bisection and Random Partition (Chen et al., 2004a) report complexities of $O(n)$ and $O(n \log n)$, respectively, they fail to preserve the failure-detection effectiveness of the original FSCS-ART (Mayer and Schneckenburger, 2006b). Chow et al. (Chow et al., 2013) proposed a *divide-and-conquer* approach by partitioning the input domain into sub-domains and using FSCS-ART to generate a *fixed* number of test cases in each sub-domain. When the number of test cases in a partition reaches a certain *threshold*, then the algorithm further partitions the sub-domains. However, determining the threshold value is a potential limitation of the method.

ARTsum (Barus et al., 2016) is another linear-order algorithm, based on category-choice partitioning (Ostrand and Balcer, 1988). However, ARTsum is mainly used for non-numeric programs, whereas our work is primarily focused on numeric programs.

7.2.3. MART

Mirror Adaptive Random Testing (MART) (Chen et al., 2004b) is an ART overhead reduction strategy that leverages the relative computational cheapness of mapping functions (compared to test case generation). One SUT input domain partition is selected as the *source* domain, where the ART method is implemented. Test cases generated in the source domain are then systematically mapped into the remaining partitions (*mirror* domains). Dynamic MART (D-MART) (Huang et al., 2015) incorporated a "divide-and-conquer" approach, but suffered from mirroring deficiency because only one mirror test case was generated at a time. E-MART (Omari et al., 2019) added flexibility to the mirror generation option of D-MART using a smart mirror allocation scheme.

Although mirror-based ART techniques are efficient, their complexities depend on the adopted ART algorithm (in the source domain), which may be quadratic — MART combined with FSCS-ART, for example, has a quadratic complexity of $O(n^2/m^2)$. Possibly because the "edge-effect" (Chen et al., 2008) in the source domain will be inherited in all mirror domains, MART failure-detection has been found to be inferior to FSCS-ART. Furthermore, preparing the initial MART testing setup, including selection of the partitioning scheme, remains an open issue. Finally, even the most recent MART enhancements (E-MART and D-MART, for example) still have an exponential time complexity relationship with the dimensions of the SUT. SWFC-ART differs from MART and other partition-based approaches: Instead of partitioning the SUT input domain, it partitions the executed test cases into a unique graph-based data structure for efficient NNS queries.

8. Conclusion and future work

FSCS-ART is a well-known ART method that has been widely researched and applied to test real-life programs of varying input types. Unfortunately, it faces severe computational issues relating to the growth in number of executed test cases and the dimensionality of the SUT's input domain: These issues have been referred to as the double-tier efficiency problem. Some existing FSCS-ART efficiency enhancement methods may only solve one-tier of the problem, by reducing time complexity related to the number of test cases generated. However, these methods can have quadratic (Section 7.1.5) or exponential (Section 7.1.4) time complexity with respect to the dimensions of the SUT. Although a method may perform well in low dimensional input domains, it may experience severe performance degradation in higher dimensions. Because real-life programs often have both many dimensions, and very low failure rates, addressing only the complexity related to the number of test cases can only solve one aspect of the efficiency problem.

Our method, SWFC-ART, comprehensively solves the FSCS-ART efficiency problems by using a hierarchical navigable small world graph (HNSWG) to store the executed, non-failure-causing test cases, thus improving the nearest neighbor search (NNS) efficiency. SWFC-ART reduces the FSCS-ART complexity from $O(n^2)$ to $O(n \cdot \log n)$, with a negligible computational impact when increasing the dimensionality. SWFC-ART preserves the original FSCS-ART failure-detection effectiveness, and delivers efficient incremental test case generation.

As this work addresses the FSCS-ART double-tier efficiency problem, there is great potential for many other interesting and promising research directions.

The primary requirement for the application of any ART method to non-numeric programs is the capability to support a suitable similarity measure (Huang et al., 2019). The underlying HNSWG structure of SWFC-ART fully supports a wide variety of numeric and non-numeric distance (similarity) metrics,[5] including: BrayCurtis dissimilarity; Canberra distance; correlation distance; cosine distance; Euclidean distance; inner product; and Manhattan distance (Deza and Deza, 2009). Non-numeric programs, including object-oriented (OO) software, can also be tested: Our proposed method can be combined with ART methods designed explicitly for OO programs, such as ARTOO (Ciupa et al., 2008). However, further research will be necessary to enable correct formulating of the framework. Empirical studies will also be needed to analyze the proposed method's performance against other non-numeric ART methods. Even for numeric programs, we will examine employing alternatives to the Euclidean distance in SWFC-ART: Such alternatives may result in better effectiveness while maintaining similar efficiency (Huang et al., 2020).

Although SWFC-ART shows similar failure-detection effectiveness to FSCS-ART, it will be interesting to examine combining it with other failure-detection enhancement methods for FSCS-ART (Li et al., 2019; Schneckenburger and Schweiggert, 2008; Kuo et al., 2008, 2007a; Chen et al., 2007c; Kuo et al., 2007b; Mayer, 2006a). Similarly, using SWFC-ART in source domains of MART (Chen et al., 2004b) may further boost its efficiency. The use of HNSWGs in ART strategies other than FSCS-ART is also worth exploring, and will also form part of our future work.

Finally, other ANNS approaches that have proven their efficiency in similarity search applications, including vector quantization (Gray and Neuhoff, 1998; Jégou et al., 2011) and hashing techniques (Indyk and Motwani, 1998), will also be examined for applicability to ART. Furthermore, boosting ANNS by exploiting the capabilities of modern hardware, such as using SIMD (Ashfaq et al., 2020) and GPUs (Johnson et al., 2019), will also be explored in our future work.

CRediT authorship contribution statement

Muhammad Ashfaq: Conceptualization, Methodology, Software, Data curation, Writing - original draft. **Rubing Huang:** Supervision, Proof-reading, Investigation, Writing - review & editing. **Dave Towey:** Writing - review & editing, Validation. **Michael Omari:** Visualization, Writing - review & editing. **Dmitry Yashunin:** Method illustrations, Methodology verification. **Patrick Kwaku Kudjo:** Formal analysis. **Tao Zhang:** Proof-reading.

[5] https://github.com/jelmerk/hnswlib.

Declaration of competing interest

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

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by the National Natural Science Foundation of China, under Grant Nos. 61872167, and 61502205.

References

Ackah-Arthur, H., Chen, J., Towey, D., Omari, M., Xi, J., Huang, R., 2019. One-domain-one-input: Adaptive random testing by orthogonal recursive bisection with restriction. IEEE Trans. Reliab. 68 (4), 1404–1428.
Agrawal, V.D., 1978. When to use random testing. IEEE Trans. Comput. 27 (11), 1054–1055.
Ammann, P., Knight, J., 1988. Data diversity: an approach to software fault tolerance. IEEE Trans. Comput. 37 (4), 418–425.
Aoyama, K., Saito, K., Sawada, H., Ueda, N., 2011. Fast approximate similarity search based on degree-reduced neighborhood graphs. In: 17th ACM International Conference on Knowledge Discovery and Data Mining. ACM Press, pp. 1055–1063.
Arcuri, A., Briand, L., 2011. Adaptive random testing: An illusion of effectiveness? In: International Symposium on Software Testing and Analysis. ACM, pp. 265–275.
Arya, S., Mount, D.M., 1993. Approximate nearest neighbor queries in fixed dimensions. In: The 4th Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, pp. 271–280.
Ashfaq, M., Huang, R., Omari, M., 2020. FSCS-SIMD: An efficient implementation of fixed-size-candidate-set adaptive random testing using SIMD instructions. In: 31st International Symposium on Software Reliability Engineering. IEEE, pp. 277–288.
Aumüller, M., Bernhardsson, E., Faithfull, A.J., 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Inf. Syst. 87, 34–49.
Barus, A.C., Chen, T.Y., Kuo, F.-C., Liu, H., Merkel, R., Rothermel, G., 2016. A cost-effective random testing method for programs with non-numeric inputs. IEEE Trans. Comput. 65 (12), 3609–3623.
Bati, H., Giakoumakis, L., Herbert, S., Surna, A., 2007. A genetic approach for random testing of database systems. In: 33rd International Conference on Very Large Data Bases. VLDB Endowment, pp. 1243–1251.
Beaumont, O., Kermarrec, A.-M., Marchal, L., Riviere, E., 2007a. VoroNet: A scalable object network based on Voronoi tessellations. In: IEEE International Parallel and Distributed Processing Symposium. IEEE, pp. 1–10.
Beaumont, O., Kermarrec, A.-M., Rivière, É., 2007b. Peer to peer multidimensional overlays: Approximating complex structures. In: Principles of Distributed Systems. Springer, pp. 315–328.
Bellman, R., 1954. The theory of dynamic programming. Bull. Amer. Math. Soc. 60 (6), 503–515.
Bellman, R., 1957. Dynamic Programming, first ed. Princeton University Press, Princeton, NJ, USA.
Bertolino, A., Cruciani, E., Miranda, B., Verdecchia, R., 2020. Know Your Neighbor: Fast Static Prediction of Test Flakiness. Technical Report, ISTI Technical Reports 2020/001.
Bin Ali, N., Engström, E., Taromirad, M., Mousavi, M.R., Minhas, N.M., Helgesson, D., Kunze, S., Varshosaz, M., 2019. On the search for industry-relevant regression testing research. Empir. Softw. Eng. 24 (4), 2020–2055.
Bishop, P., 1993. The variation of software survival time for different operational input profiles. In: 23rd International Symposium on Fault-Tolerant Computing. IEEE Computer Society, pp. 98–107.
Branicky, M., LaValle, S.M., Olson, K., Libo Yang, 2001. Quasi-randomized path planning. In: IEEE International Conference on Robotics and Automation (Cat. No.01CH37164), vol. 2. IEEE, pp. 1481–1487.
Bueno, P.M.S., Jino, M., Wong, W.E., 2014. Diversity oriented test data generation using metaheuristic search techniques. Inform. Sci. 259, 490–509.
Caretta Cartozo, C., De Los Rios, P., 2009. Extended navigability of small world networks: exact results and new insights. Phys. Rev. Lett. 102 (23), 238703:1–238703:4.
Cartaxo, E.G., Machado, P.D., Neto, F.O., 2011. On the use of a similarity function for test case selection in the context of model-based testing. Softw. Test. Verif. Reliab. 21 (2), 75–100.
Chan, F., Chan, K.P., Chen, T.Y., Yiu, S.M., 2004. Adaptive random testing with CG constraint. In: 28th Annual International Computer Software and Applications Conference. 2, IEEE Computer Society, pp. 96–99.
Chan, F.T., Chen, T.Y., Mak, I.K., Yu, Y.T., 1996. Proportional sampling strategy: Guidelines for software testing practitioners. Inf. Softw. Technol. 38 (12), 775–782.
Chan, K.P., Chen, T.Y., Towey, D., 2006. Forgetting test cases. In: 30th Annual International Computer Software and Applications Conference, vol. 1. IEEE, pp. 485–494.
Chávez, E., Tellez, E.S., 2010. Navigating k-nearest neighbor graphs to solve nearest neighbor searches. In: Mexican Conference on Pattern Recognition. Springer, pp. 270–280.
Chen, J., Ackah-Arthur, H., Kudjo, P.K., Mao, C., 2019. A taxonomic review of adaptive random testing for numeric programs. ArXiv Preprint ArXiv: 1909.10879.
Chen, T.Y., Huang, D.H., Kuo, F.-C., 2007c. Adaptive random testing by balancing, in: 2nd International Workshop on Random Testing Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering, ACM Press, pp. 2–9.
Chen, T.Y., Kuo, F.-C., Liu, H., 2007b. On test case distributions of adaptive random testing. In: 19th International Conference on Software Engineering and Knowledge Engineering. Knowledge Systems Institute–Graduate School, pp. 141–144.
Chen, T.Y., Kuo, F.-C., Liu, H., 2008. Distributing test cases more evenly in adaptive random testing. J. Syst. Softw. 81 (12), 2146–2162.
Chen, T., Kuo, F.-C., Liu, H., Wong, W.E., 2008. Does adaptive random testing deliver a higher confidence than random testing? In: The 8th International Conference on Quality Software. IEEE, pp. 145–154.
Chen, T.Y., Kuo, F.-C., Liu, H., Wong, W.E., 2013. Code coverage of adaptive random testing. IEEE Trans. Reliab. 62 (1), 226–237.
Chen, T., Kuo, F.-C., Liu, H., Wong, W.E., 2013. Code coverage of adaptive random testing. IEEE Trans. Reliab. 62 (1), 226–237.
Chen, T.Y., Kuo, F.-C., Merkel, R., 2006b. On the statistical properties of testing effectiveness measures. J. Syst. Softw. 79 (5), 591–601.
Chen, T.Y., Kuo, F.-C., Merkel, R.G., Ng, S.P., 2004b. Mirror adaptive random testing. Inf. Softw. Technol. 46 (15), 1001–1010.
Chen, T.Y., Kuo, F.-C., Merkel, R.G., Tse, T.H., 2010. Adaptive random testing: The ART of test case diversity. J. Syst. Softw. 83 (1), 60–66.
Chen, T.Y., Kuo, F.-C., Zhou, Z.Q., 2007a. On favourable conditions for adaptive random testing. Int. J. Softw. Eng. Knowl. Eng. 17 (06), 805–825.
Chen, T.Y., Leung, H., Mak, I.K., 2005. Adaptive random testing. In: Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making. Springer, pp. 320–329.
Chen, T.Y., Merkel, R., 2006a. Efficient and effective random testing using the Voronoi diagram. In: 17th Australian Software Engineering Conference. IEEE Computer Society, pp. 300–305.
Chen, T.Y., Merkel, R., 2008. An upper bound on software testing effectiveness. ACM Trans. Softw. Eng. Methodol. 17 (3), 16:1–16:27.
Chen, T.Y., Merkel, R., Wong, P., Eddy, G., 2004a. Adaptive random testing through dynamic partitioning. In: 4th International Conference on Quality Software. IEEE Computer Society, pp. 79–86.
Chow, C., Chen, T.Y., Tse, T.H., 2013. The ART of divide and conquer: An innovative approach to improving the efficiency of adaptive random testing. In: 13th International Conference on Quality Software. IEEE, pp. 268–275.
Ciupa, I., Leitner, A., Oriol, M., Meyer, B., 2008. ARTOO: adaptive random testing for object-oriented software. In: The 30th International Conference on Software Engineering. ACM, pp. 71–80.
Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences, second ed. Hillsdale, N.J.: L. Erlbaum Associates.
Collected Algorithms of the ACM, 2020. http://calgo.acm.org/ (Accessed 14 January 2020).
Cruciani, E., Miranda, B., Verdecchia, R., Bertolino, A., 2019. Scalable approaches for test suite reduction. In: The IEEE/ACM 41st International Conference on Software Engineering. IEEE, pp. 419–429.
Daboczi, T., Kollar, I., Simon, G., Megyeri, T., 2003. Automatic testing of graphical user interfaces. In: 20th IEEE Instrumentation Technology Conference, vol. 1. IEEE, pp. 441–445.
Devroey, X., Perrouin, G., Legay, A., Schobbens, P.-Y., Heymans, P., 2016. Search-based similarity-driven behavioural SPL testing. In: The 10th International Workshop on Variability Modelling of Software-Intensive Systems. ACM, pp. 89–96.
Deza, M.M., Deza, E., 2009. Encyclopedia of Distances. Springer Berlin Heidelberg.
Do, H., Elbaum, S., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10 (4), 405–435.
Domingos, P., 2012. A few useful things to know about machine learning. Commun. ACM 55 (10), 78–87.
Dong, W., Moses, C., Li, K., 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In: 20th International Conference on World Wide Web. ACM Press, pp. 577–586.
Duran, J.W., Ntafos, S., 1981. A report on random testing. In: 5th International Conference on Software Engineering, vol. 81. IEEE, pp. 179–183.
Duran, J., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Trans. Softw. Eng. 10 (4), 438–444.

Ferrer, J., Chicano, F., Alba, E., 2012. Evolutionary algorithms for the multi-objective test data generation problem. Softw. - Pract. Exp. 42 (11), 1331–1362.
Finelli, G.B., 1991. NASA software failure characterization experiments. Reliab. Eng. Syst. Saf. 32 (1–2), 155–169.
Forrester, J.E., Miller, B.P., 2000. An empirical study of the robustness of windows NT applications using random testing. In: 4th Conference on USENIX Windows Systems Symposium, vol. 4. USENIX Association, pp. 59–68.
Freund, J.E., 1988. Modern Elementary Statistics. Prentice-Hall, Inc..
Giorgetti, A., Marché, C., Tushkanova, E., Kouchnarenko, O., 2010. Specifying generic Java programs: Two case studies, in: 10th Workshop on Language Descriptions, Tools and Applications, ACM, pp. 8:1–8:8.
Godefroid, P., Kiezun, A., Levin, M.Y., 2008. Grammar-based whitebox fuzzing. In: 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp. 206–215.
Gray, R., Neuhoff, D.L., 1998. Quantization. IEEE Trans. Inform. Theory 44 (6), 2325–2383.
Hajebi, K., Abbasi-Yadkori, Y., Shahbazi, H., Zhang, H., 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In: 22nd International Joint Conference on Artificial Intelligence. 2, AAAI Press, pp. 1312–1317.
Hamlet, D., 2002. Continuity in software systems. SIGSOFT Softw. Eng. Notes 27 (4), 196–200.
Hamlet, R., Maciniak, J., 1994. Random Testing. Encyclopedia of Software Engineering. Wiley, New York, pp. 970–978.
Huang, R., Cui, C., Sun, W., Towey, D., 2020. Poster: Is euclidean distance the best distance measurement for adaptive random testing? In: 13th International Conference on Software Testing, Validation and Verification. IEEE, pp. 406–409.
Huang, R., Liu, H., Xie, X., Chen, J., 2015. Enhancing mirror adaptive random testing through dynamic partitioning. Inf. Softw. Technol. 67, 13–29.
Huang, R., Sun, W., Xu, Y., Chen, H., Towey, D., Xia, X., 2019. A survey on adaptive random testing. IEEE Trans. Softw. Eng. 1–35.
Indyk, P., Motwani, R., 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In: The 30th Annual ACM Symposium on Theory of Computing. ACM, pp. 604–613.
ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary, 2010. ISO/IEC/IEEE 24765:2010(E), pp. 1–418.
Java API Documentation, 2020. https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html (Accessed 12 December 2020).
Jégou, H., Douze, M., Schmid, C., 2011. Product quantization for nearest neighbor search. IEEE Trans. Pattern Anal. Mach. Intell. 33 (1), 117–128.
JetBrains Academy, 2020. https://hyperskill.org/projects/60?track=1 (Accessed 12 December 2020).
Jia, Y., Harman, M., 2010. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.
Jiang, Z., Xie, L., Deng, X., Xu, W., Wang, J., 2016. Fast nearest neighbor search in the Hamming space. In: International Conference on Multimedia Modeling. Springer, pp. 325–336.
Jixin Geng, Jiongmin Zhang, 2010. A new method to solve the "boundary effect" of adaptive random testing. In: 2010 International Conference on Educational and Information Technology, vol. 1. IEEE, pp. 298–302.
Johnson, J., Douze, M., Jégou, H., 2019. Billion-scale similarity search with GPUs. IEEE Trans. Big Data 1–14.
Karbasi, A., Ioannidis, S., Massoulie, L., 2015. From small-world networks to comparison-based search. IEEE Trans. Inform. Theory 61 (6), 3056–3074.
Kleinberg, J.M., 2000a. Navigation in a small world. Nature 406, 845.
Kleinberg, J., 2000b. The small-world phenomenon: An algorithmic perspective. In: 32nd Annual ACM Symposium on Theory of Computing. ACM, pp. 163–170.
Kuo, F.-C., Chen, T.Y., Liu, H., Chan, W.K., 2007a. Enhancing adaptive random testing in high dimensional input domains. In: ACM Symposium on Applied Computing. ACM, pp. 1467–1472.
Kuo, F.-C., Chen, T.Y., Liu, H., Chan, W.K., 2008. Enhancing adaptive random testing for programs with high dimensional input domains or failure-unrelated parameters. Softw. Qual. J. 16 (3), 303–327.
Kuo, F.-C., Sim, K.Y., Sun, C., Tang, S., Zhou, Z., 2007b. Enhanced random testing for programs with high dimensional input domains. In: International Conference on Software Engineering and Knowledge Engineering. Knowledge Systems Institute, pp. 135–140.
Lee, D.-T., Wong, C., 1977. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. Acta Inform. 9 (1), 23–29.
Li, Z., Li, Q., Li, R., Wang, L., 2019. An enhanced ART in high dimensional input domain. In: 10th International Conference on Software Engineering and Service Science. IEEE, pp. 495–497.
Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., Lin, X., 2019. Approximate nearest neighbor search on high dimensional data-experiments, analyses, and improvement. IEEE Trans. Knowl. Data Eng. 32 (8), 1475–1488.
Liang, Y.D., 2017. Introduction To Java Programming and Data Structures, Comprehensive Version. Pearson Education.
Lifshits, Y., Zhang, S., 2009. Combinatorial algorithms for nearest neighbors, near-duplicates and small-world design. In: 20th Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, pp. 318–326.
Lin, Y., Tang, X., Chen, Y., Zhao, J., 2009. A divergence-oriented approach to adaptive random testing of Java programs. In: 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 221–232.
Liu, H., Xie, X., Yang, J., Lu, Y., Chen, T.Y., 2010. Adaptive random testing by exclusion through test profile. In: 10th International Conference on Quality Software. IEEE, pp. 92–101.
Liu, H., Xie, X., Yang, J., Lu, Y., Chen, T.Y., 2011. Adaptive random testing through test profiles. Softw. - Pract. Exp. 41 (10), 1131–1154.
Lv, C., Zhang, L., Zeng, F., Zhang, J., 2019. Adaptive random testing for XSS vulnerability. In: The 26th Asia-Pacific Software Engineering Conference. IEEE, pp. 63–69.
Malkov, Y., Ponomarenko, A., 2016. Growing homophilic networks are natural navigable small worlds. PLoS One 11 (6), e0158162:1–e0158162:14.
Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V., 2012. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In: International Conference on Similarity Search and Applications. Springer, pp. 132–147.
Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V., 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. Inf. Syst. 45, 61–68.
Malkov, Y., Yashunin, D.A., 2020. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. Pattern Anal. Mach. Intell. 42 (4), 824–836.
Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. 50–60.
Mao, C., Chen, T.Y., Kuo, F.-C., 2017. Out of sight, out of mind: A distance-aware forgetting strategy for adaptive random testing. Sci. China Inf. Sci. 60 (9), 1–21.
Mao, C., Zhan, X., Tse, T.H., Chen, T.Y., 2019. KDFC-ART: a KD-tree approach to enhancing fixed-size-candidate-set adaptive random testing. IEEE Trans. Reliab. 68 (4), 1444–1469.
May, P.S., 2007. Test Data Generation: Two Evolutionary Approaches to Mutation Testing (Ph.D. thesis). Computing Laboratory, The University of Kent.
Mayer, J., 2006a. Towards effective adaptive random testing for higher-dimensional input domains. In: 8th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1955–1956.
Mayer, J., Schneckenburger, C., 2006. Statistical analysis and enhancement of random testing methods also under constrained resources. In: International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, vol. 1. CSREA Press, pp. 16–23.
Mayer, J., Schneckenburger, C., 2006b. An empirical analysis and comparison of random testing techniques. In: 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ACM, pp. 105–114.
Mehlhorn, H., Schreiber, F., 2013. Small-world property. In: Encyclopedia of Systems Biology. Springer, pp. 1957–1959.
Milgram, S., 1967. The small world problem. Psychol. Today 2 (1), 60–67.
Miller, B.P., Cooksey, G., Moore, F., 2006. An empirical study of the robustness of MacOS applications using random testing. In: 1st International Workshop on Random Testing. ACM, pp. 46–54.
Miller, B.P., Fredriksen, L., So, B., 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM 33 (12), 32–44.
Miller, B.P., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J., 1995. Fuzz Revisited: a Re-Examination of the Reliability of UNIX Utilities and Services. Technical Report, University of Wisconsin–Madison Department of Computer Sciences.
Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A., 2018. FAST approaches to scalable similarity-based test case prioritization. In: The 2018 IEEE/ACM 40th International Conference on Software Engineering. ACM, pp. 222–232.
Muangsiri, W., Takada, S., 2017. Random GUI testing of Android application using behavioral model. Int. J. Softw. Eng. Knowl. Eng. 27 (09–10), 1603–1612.
Myers, G.J., 1979. The Art of Software Testing. Wiley, New York.
Omari, M., Chen, J., Ackah-Arthur, H., Kudjo, P.K., 2019. Elimination by linear association: An effective and efficient static mirror adaptive random testing. IEEE Access 7, 71038–71060.
O'Neil, D.J., 2017. Nearest neighbor problem. In: Encyclopedia of GIS. Springer International Publishing, Cham, pp. 1421–1426.
Ostrand, T.J., Balcer, M.J., 1988. The category-partition method for specifying and generating functional tests. Commun. ACM 31 (6), 676–686.
Pacheco, C., Lahiri, S.K., Ball, T., 2008. Finding errors in .NET with feedback-directed random testing. In: 2008 International Symposium on Software Testing and Analysis. ACM, pp. 87–96.
Pallant, J., 2016. SPSS Survival Manual: A Step By Step Guide To Data Analysis using SPSS. Open University Press.
Paredes, R., 2008. Graphs for metric space searching (Ph.D. thesis). University of Chile, Chile, Dept. of Computer Science Tech Report TR/DCC-2008-10.

Ponomarenko, A., Avrelin, N., Naidan, B., Boytsov, L., 2014. Comparative analysis of data structures for approximate nearest neighbor search. Data Anal. 125–130.
Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., 2007. Numerical Recipes: The Art of Scientific Computing, third ed. Cambridge University Press, USA.
Pugh, W., 1990. Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33 (6), 668–676.
Regehr, J., 2005. Random testing of interrupt-driven software. In: 5th ACM International Conference on Embedded Software. ACM, pp. 290–298.
Schneckenburger, C., Mayer, J., 2007. Towards the determination of typical failure patterns. In: 4th International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting. ACM, pp. 90–93.
Schneckenburger, C., Schweiggert, F., 2008. Investigating the dimensionality problem of Adaptive Random Testing incorporating a local search technique, in: 1st International Conference on Software Testing Verification and Validation Workshop, IEEE Computer Society, pp. 241–250.
Shahbazi, A., Tappenden, A.F., Miller, J., 2013. Centroidal voronoi tessellations–a new approach to random testing. IEEE Trans. Softw. Eng. 39 (2), 163–183.
Slutz, D.R., 1998. Massive stochastic testing of SQL. In: 24th International Conference on Very Large Data Bases. Morgan Kaufmann Publishers, pp. 618–622.
Sullivan, G.M., Feinn, R., 2012. Using effect size—or why the *p* value is not enough. J. Graduate Med. Educ. 4 (3), 279–282.
Walkinshaw, N., Fraser, G., 2017. Uncertainty-driven black-box test data generation. In: International Conference on Software Testing, Verification and Validation. IEEE Computer Society, pp. 253–263.
Wang, J., Li, S., 2012. Query-driven iterated neighborhood graph search for large scale indexing. In: 20th ACM International Conference on Multimedia. ACM Press, pp. 179–188.
Wang, J., Wang, J., Zeng, G., Gan, R., Li, S., Guo, B., 2015. Fast neighborhood graph search using cartesian concatenation. In: Multimedia Data Mining and Analytics - Disruptive Innovation. Springer, pp. 397–417.
Watts, D., Strogatz, S., 1998. Collective dynamics of 'small-world' networks. Nature 393 (6684), 440–442.
White, L., Cohen, E., 1980. A domain strategy for computer program testing. IEEE Trans. Softw. Eng. 6 (3), 247–257.
Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biom. Bull. 1 (6), 80–83.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2012. Experimentation in software engineering. Springer.
Yan, M., Wang, L., Fei, A., 2020. ARTDL: Adaptive random testing for deep learning systems. IEEE Access 8, 3055–3064.
Yoshikawa, T., Shimura, K., Ozawa, T., 2003. Random program generator for Java JIT compiler test system. In: 3rd International Conference on Quality Software. IEEE, pp. 20–23.
Zhang, L., Zhang, D., Wang, C., Zhao, J., Zhang, Z., 2019. ART4sqli: The ART of SQL injection vulnerability discovery. IEEE Trans. Reliab. 68 (4), 1470–1489.

Muhammad Ashfaq received the Bachelor's degree in Information Technology in 2017 from University of Gujrat, Pakistan. Currently he is pursuing Master's degree in Computer Science and Technology from the School of Computer Science and Communication Engineering, Jiangsu University, China. His current research interests include software testing and software debugging. His work has been published in International Symposium on Software Reliability Engineering (ISSRE 2020). One of his works has been accepted to be presented in the proceedings of 12th Asia-Pacific Symposium on Internetware (Internetware 2020).

Rubing Huang received the Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2013. From 2016 to 2018, he was a visiting scholar at Swinburne University of Technology and at Monash University, Australia. He is an associate professor at the Faculty of Information and Technology, Macau University of Science and Technology (MUST). Before joining MUST, he worked as an associate professor at Jiangsu University, China. His current research interests include software testing (including adaptive random testing, random testing, failure-based testing, combinatorial testing, and regression testing), debugging, and maintenance. He has more than 50 publications in journals and proceedings, including in IEEE Transactions on Software Engineering, IEEE Transactions on Reliability, IEEE Transactions on Emerging Topics in Computational Intelligence, Journal of Systems and Software, Information and Software Technology, IET Software, The Computer Journal, International Journal of Software Engineering and Knowledge Engineering, Information Sciences, ICSE, ISSRE, ICST, COMPSAC, QRS, SEKE, and SAC. He is a senior member of the IEEE and the China Computer Federation, and a member of the ACM. More information about him and his work is available online at https://huangrubing.github.io/.

Dave Towey received the B.A. and M.A. degrees in computer science, linguistics, and languages from the University of Dublin, Trinity College, Ireland; the M.Ed. degree in education leadership from the University of Bristol, U.K.; and the Ph.D. degree in computer science from The University of Hong Kong, China. He is an associate professor at University of Nottingham Ningbo China (UNNC), in Zhejiang, China, where he serves as the deputy head of the School of Computer Science. He is also the deputy director of the International Doctoral Innovation Center. He is a member of the UNNC Artificial Intelligence and Optimization research group. His current research interests include software testing (especially adaptive random testing, for which he was amongst the earliest researchers who established the field, and metamorphic testing), computer security, and technology-enhanced education. He co-founded the ICSE International Workshop on Metamorphic Testing in 2016. He is a fellow of the HEA, a senior member of the IEEE, and a member of the ACM.

Michael Omari received the B.Sc. degree in computer science from the University of Ghana, Legon, in 2007, and the Master's degree in information technology from Coventry University, U.K., in 2014. He holds a Ph.D. degree in computer applied technology from the School of Computer Science and Telecommunication Engineering, Jiangsu University, China. He is a lecturer at the Department of Computer Science, School of Applied Science, Takoradi Technical University specializing in assembly language programming and software engineering. His research interests include software testing and embedded systems.

Dmitry Yashunin received a Master's degree in physics from Nizhny Novgorod State University in 2009, and a Ph.D. degree in laser physics from the Institute of Applied Physics RAS in 2015. From 2008 to 2012 he was working at Mera Networks (currently Orion Innovation) as a software engineer. From 2016 to 2019 he was working at Intelli-Vision in the position of a leading research engineer. Dmitry currently works at Harman (a Samsung company) as an associate director. He is author of more than 10 papers on physics and computer science. His current research interests include scalable similarity search, computer vision and deep learning.

Patrick Kwaku Kudjo is a Lecturer at the Department of Information Technology, University of Professional Studies, Ghana. He has a Ph.D. in Computer Application Technology awarded by Jiangsu University, China. He holds a Master's degree in Information Technology from Sikkim Manipal University, India, and a Bachelor's degree in Computer Science and Management from Wisconsin International University College, Accra Ghana. Patrick is an avid researcher and practitioner with more than twenty high-quality research papers in reputable journals and conferences such as Journal of Systems and Software, Software Quality Journal, Software: Practice and Experience, Applied Intelligence, Service-Oriented Computing, and Applications ISSTA, QRS, ASE, SEKE, etc. His current research interest includes Information Security, Machine Learning, and Blockchain Analysis. He is a member of the Association for Computing Machinery (ACM), and the Institute of Electrical and Electronics Engineers (IEEE).

Tao Zhang received the BS degree in automation, the MEng degree in software engineering from Northeastern University, China, and the Ph.D. degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Currently, he is an associate professor with the Faculty of Information Technology, Macau University of Science and Technology (MUST). Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 50 high-quality papers at renowned software engineering and security journals and conferences such as the IEEE Transactions on Software Engineering, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, IEEE Software, ICSE, etc. His current research interests include mining software repositories and mobile software security. He is a senior member of ACM, IEEE, and CCF.