symmetry

MDPI

Article

DroidbotX: Test Case Generation Tool for Android Applications Using Q-Learning

Husam N. Yasin *, Siti Hafizah Ab Hamid * and Raja Jamilah Raja Yusof *

Department of Software Engineering, Faculty of Computer Science and Information Technology, Universiti Malaya, Kuala Lumpur 50603, Malaysia

* Correspondence: husam_yas@siswa.um.edu.my (H.N.Y.); sitihafizah@um.edu.my (S.H.A.H.); rjry@um.edu.my (R.J.R.Y.)

Abstract: Android applications provide benefits to mobile phone users by offering operative functionalities and interactive user interfaces. However, application crashes give users an unsatisfactory experience, and negatively impact the application's overall rating. Android application crashes can be avoided through intensive and extensive testing. In the related literature, the graphical user interface (GUI) test generation tools focus on generating tests and exploring application functions using different approaches. Such tools must choose not only which user interface element to interact with, but also which type of action to be performed, in order to increase the percentage of code coverage and to detect faults with a limited time budget. However, a common limitation in the tools is the low code coverage because of their inability to find the right combination of actions that can drive the application into new and important states. A Q-Learning-based test coverage approach developed in DroidbotX was proposed to generate GUI test cases for Android applications to maximize instruction coverage, method coverage, and activity coverage. The overall performance of the proposed solution was compared to five state-of-the-art test generation tools on 30 Android applications. The DroidbotX test coverage approach achieved 51.5% accuracy for instruction coverage, 57% for method coverage, and 86.5% for activity coverage. It triggered 18 crashes within the time limit and shortest event sequence length compared to the other tools. The results demonstrated that the adaptation of Q-Learning with upper confidence bound (UCB) exploration outperforms other existing state-of-the-art solutions.

Keywords: android; GUI Testing; test case generation; reinforcement learning; Q-learning

Citation: Yasin, H.N.; Hamid, S.H.A.; Raja Yusof, R.J. DroidbotX: Test Case Generation Tool for Android Applications Using Q-Learning. *Symmetry* **2021**, *13*, 310. https://doi.org/10.3390/sym13020310

Academic Editor: José Carlos R. Alcantud

Received: 29 December 2020
Accepted: 10 February 2021
Published: 12 February 2021

1. Introduction

Android operates on 85% of mobile phones with over 2 billion active devices per month worldwide [1]. The Google Play Store is the official market for Android applications (apps) that distribute over 3 million Android apps in 30 categories. For example, it provides entertainment, customization, education, and financial apps [2]. A previous study [3] indicated that a mobile device, on average, has between 60 and 90 apps installed. Besides, an Android user, on average spends 2 h and 15 min on apps every day. Therefore, checking the app's reliability is a significant task. Recent research [2] showed that the number of Android apps downloaded is increasing drastically every year. Unfortunately, 17% of Android apps were still considered to be low-quality apps in 2019 [4]. Another study found that 53% of users would avoid using an app if the app crashed [5]. A mobile app crash not only offers a poor user experience but also negatively impact the app's overall rating [6,7]. The inferior quality of Android apps can be attributed to insufficient testing due to its rapid development practice. Android apps are ubiquitous, operating in complex environments, and evolve under market pressure. Android developers neglect appropriate testing practices as they consider it time-consuming, expensive, and involving a lot of repetitive tasks. Mobile app crashes are evitable and avoidable by intensive and extensive testing of mobile apps [6]. Through a graphical user interface (GUI), mobile

app testing verifies the functionality and accuracy of mobile apps before these apps are released to the market [8–10]. Automated mobile app testing starts by generating test cases that include event sequences of the GUI components. In the mobile app environment, the test input (or test data) will be based on user interaction and system interaction (e.g., apps notification). The development of GUI test cases usually takes a lot of time and effort because of their non-trivial structures and highly interactive nature of GUIs. Android apps [11,12] usually possess many states and transitions, which can lead to an arduous testing process and poor testing performance for large apps. Over the past decade, Android test generation tools have been developed to automate user interaction and system interaction as inputs [13–18]. The purpose of these tools is to generate test cases and explore the app's functions by employing different techniques. These techniques are random-based, model-based, systematic based, and reinforcement learning. However, there are issues with low code coverage of existing tools [11,19,20], due to the inability to explore app functions extensively because some of the app functions can only be explored through a specific sequence of events [21]. Such tools must not only choose which GUI component to interact with but also which type of input to perform. Each type of input for each GUI component is likely to improve coverage. Coverage is an important metric to measure the efficiency of testing [22]. Combining different granularities from instruction, method, and activity coverage is beneficial for better results in testing Android apps. The reason is that activities and methods are vital to app development, so the numeric values of activity and method coverage are intuitive and informative [23]. Activity is the primary interface for user interaction and an activity comprises several methods and underlying code logic. Each method in every activity comprises a different number of lines of code. Instruction coverage provides information about the amount of code that has been executed. Hence, improving instruction and method coverage ensures that more of the app's functionalities associated with each activity are explored and tested [23–25]. Similarly, activity coverage is a necessary condition to detect crashes that can occur when interacting with the app's UI. The more coverage the tool explores, the more likely it would discover potential crashes [26].

This research proposes an approach that generates a GUI test case based on the Q-Learning technique. This approach systematically selects events and guides the exploration to expose the functionalities of an application under test (AUT) to maximize instruction, method, and activity coverage by minimizing redundant execution of events.

This approach was implemented into the test tool named DroidbotX (https://github.com/husam88/DroidbotX, accessed on 9 February 2021) and it is publicly available. The problem-based learning approach in teaching the public using DroidbotX is also available in the ALIEN (Active Learning in Engineering) (https://virtual-campus.eu/alien/problems/droidbotx-gui-testing-tool/, accessed on 9 February 2021) virtual platform. The tool was used to evaluate the practical usefulness and applicability of our approach. DroidbotX constructs a state-transition model of an app and generates test cases. These test cases follow the sequences of events that are the most likely to explore the app's functionalities. The proposed approach was evaluated against state-of-the-art test generation tools. DroidbotX was compared with Android Monkey [27], Sapienz [16], Stoat [15], Droidbot [28], Humanoid [29] on 30 Android apps from the F-Droid repository [30].

In this study, instruction coverage, method coverage, activity coverage, and crash detection were analyzed to assess the performance of the approach. DroidbotX achieved higher instruction coverage, method coverage, activity coverage, and detected more crashes than the other tools on the 30 subject apps. Specifically, DroidbotX consistently resulted in 51.5% instruction coverage, 57% method coverage, 86.5% activity coverage, and triggered 18 crashes over the five tools.

The rest of this paper is divided as follows. Section 2 describes a test case generation for Android apps. Section 3 discusses reinforcement learning and focused on Q-Learning. Section 4 presents the background of Android apps and Section 5 discusses the related GUI testing tools. Section 6 presents the proposed approach while Section 7 presents an

empirical evaluation. Section 8 analyzes and discusses the findings. Section 9 describes threats to validity and Section 10 concludes the paper.

2. Test Case Generation for Android Apps

Test case generation is one of the most attention-demanding testing activities because of its strong impact on the overall testing process efficiency [31]. The total cost, time, and effort required for the overall testing will depend on the total number of test cases [32]. The pre-specified test case is a set of inputs provided to the application to obtain the desired output. Android apps are context-conscious because of their ability to sense and react with a great number of different inputs from user and system interactions [33,34]. An app is tested with an automatically generated sequence of events simulating user interaction with the GUI from the user's perspective to persistence layers. For example, interaction usually involves clicking, scrolling, or typing texts into a GUI element, such as a button, image, or text block. Android apps can sense and respond to multiple inputs from system interactions [33]. Interaction with system events includes SMS notifications, app notifications, or events coming from sensors. The underlying software responds by the execution of an event handler, i.e., an ActionListener, in one of several ways. These experiences are some of the events that need to be addressed in testing Android apps, as they effectively increase the complexity of app testing [35].

3. Q-Learning

Q-learning is a type of model-free technique of reinforcement learning (RL) [36]. RL is a branch of machine learning. Unlike other branches like supervised and unsupervised learning, its algorithms are trained using reward and punishment to interact with the environment. It is based on the concept of behavioral psychology that works on interacting directly with an environment which plays a key component in artificial intelligence. In RL techniques, a reward is observed if the agent reaches an objective. RL techniques include Actor-critic, Deep Q Network (DQN), State-Action-Reward-State-Action (SARSA), and Q-Learning. The major components of RL are the agent and the environment. The agent serves as an independent entity that performs unconstrained actions within the environment in order to achieve a specific goal. The agent performs an activity on the environment and uses trial-and-error interactions to gain information about the environment. There are four other basic concepts in the RL system along with the agent and the environment: (i) policy, (ii) reward, (iii) action, and (iv) state. The state describes the present situation of the environment and mimics the behavior of the environment. For example, this gives rise to the current situation and action. The model might predict the resultant next state and the next reward. Models are used to plan and decide on a course of action by considering possible future situations before they are experienced. Similarly, the reward is an abstract concept to evaluate actions. Reward refers to immediate feedback after performing an action. The policy defines the agent approach to select an action from a given state. It is the core of the RL agent and sufficient to determine behavior. In general, policies may be stochastic. An action is a possible move in a particular state.

Q-Learning is used to find an optimal action-selection policy for the given AUT, where the policy sets out the rule that the agent must follow when choosing a particular action from a set of actions [37]. There is an action execution that is immediately preceded to choose each action, which moves the agent from the current state to a new state. This agent is rewarded with a reward r upon executing the action a. The value of the reward is then measured using the reward function R. For the agent, the main aim of Q-Learning is to learn how to act in an optimal way that maximizes the cumulative reward. Thus, a reward is granted when an entire sequence of actions is carried out.

Q-Learning uses its Q-values to resolve RL problems. For each policy Π, the action-value function or quality function (Q-function) should be properly defined. Nonetheless, the value $Q\ \Pi\ (s_t;\ a_t)$ is the expected cumulative reward that can be achieved by executing a sequence of actions that starts with action a_t from s_t; and then follows the policy Π. The

optimal Q-function Q^* is the maximum Q expected cumulative reward achievable for a given (state, action) pair over all possible policies.

$$Q^*(s_t, a_t) = \max_{\pi} \sum_{t>0} (\gamma^t r_t | s = s_t,\ a = a_t,\ \pi) \quad (1)$$

Intuitively, if Q^* is known, the optimal strategy at each step s_t is to take action that maximizes the sum: $r\ +\ Q*\ (s_t + 1,\ a_t + 1)$, where r is the immediate reward of the current step, while t stands for the current time step, hence $t\ + 1$ denotes the next one. The discount value (γ) is introduced to control the long-term rewards' relevance with the immediate one.

Figure 1 presents the RL mechanism in the context of the Android app testing. In automated GUI testing, AUT is the environment; the state is the set of actions available on the AUT screen. The GUI actions are the set of actions available in the current state of the environment, and the testing tool is the agent. Initially, the testing tool does not know the AUT. As the tool generates and executes test event input based on trial-and-error interaction, the knowledge about AUT is updated to find a policy that facilitates systematic exploration to make efficient future action selection decisions. This exploration generates event sequences that can be used as test cases.

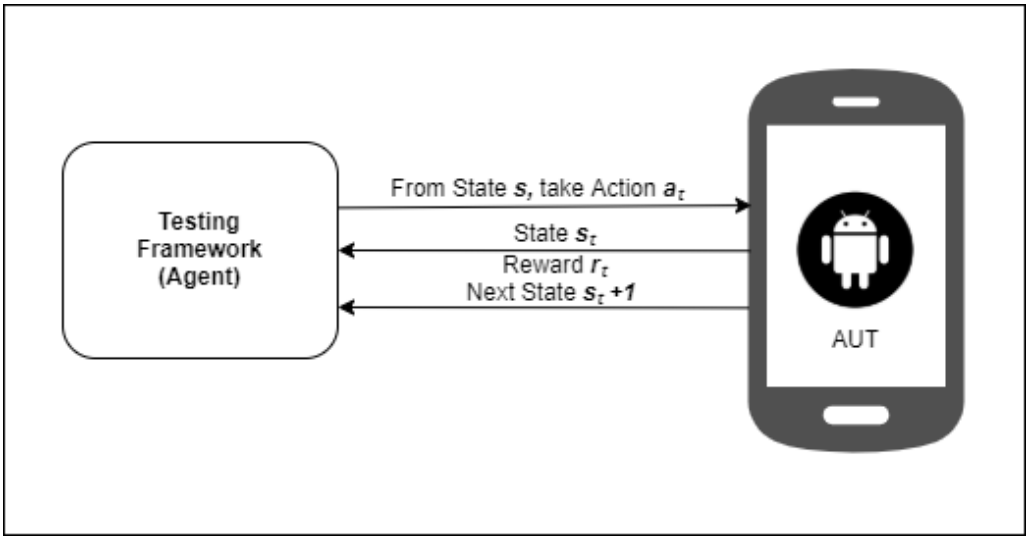

Figure 1. Reinforcement learning mechanism.

4. Android Apps Execution

There are four key components of an Android app as follows: (i) activities, (ii) services, (iii) broadcast receivers, and (iv) content providers. Each component represents a point where the user or system communicates with the GUI components. These components must be declared in the corresponding XML (eXtensible Markup Language) file. Android app manifest is an invaluable XML file stored in the root directory of the app's source as AndroidManifest.xml. When the device is compiled, the manifest file will be converted into a binary format. This file provides the necessary details about the device to the Android system, such as the package name and App ID, the minimum level of API (application programming interface) required, the list of mandatory permissions, and the hardware specifications.

Activity is the interface layer of the application the user manipulates to engage. Each activity represents a group of layouts such as the linear layout, which horizontally or vertically organizes the screen items. The interface includes GUI elements, known as widgets or controls. These elements are buttons, text boxes, search bars, switches, and number pickers. These elements allow users to interact with the apps. The widgets are handled as task stacks within the system. When an app is launched in the Android system,

a new activity starts by default. It is usually positioned at the peak of the current stack and automatically becomes the running activity. Furthermore, the previous activity then remains in the stack just below it and does not come back to the foreground until the new activity exits. Stacks of operation can be seen on the screen. Activity is the primary target of testing tools for the Android app as the user navigates through the screen. The complete lifecycle of an activity is described by the following Activity methods; created, paused, resumed, and destroyed. These methods are linked together to disable in case the activity changes status. The activity lifecycle is tightly coupled with the Android framework, which is managed by an essential service called the Activity manager [38].

The activity comprises a set of views and fragments that present information to the user while interacting with the application. A fragment is a class that contains a portion of the user interface or behavior of the app, which can be placed as part of an activity. Fragments support more dynamic and flexible user interface (UI) designs on a large screen such as tablets. It was implemented in Android from API level 11 onwards. The fragment must always be embedded in an activity, and the fragment's lifecycle is directly affected by the lifecycle of the host activity. Fragments inside the activity will be stopped if the activity is stopped and destroyed if the activity is destroyed.

5. Related Works

Researchers have developed approaches to automate test generation for Android apps. This section highlights the existing tools with corresponding approaches. Table 1 classifies Android test generation tools based on seven features as follows: (1) technique, (2) test case generation approach, (3) test inputs, (4) testing environment, (5) test artifacts, (6) basis, and (7) availability.

5.1. Automated Graphical User Interface (GUI) Testing with Q-Learning

Mariani et al. [39] proposed AutoBlackTest, the first Q-Learning-based GUI testing tool for Java desktop software. AutoBlackTest initially extracts an abstract representation of the current state of the GUI and generates a behavioral model. This model is updated according to the current state reached and the immediate utility of the action. Then the behavioral model is used to select the next action to be executed, and then to restart the loop. TESTAR [40], another Q-Learning-based tool, is used to generate GUI test sequences based on web applications. The Q-Learning algorithm provided significant performance with an adequate set of parameters.

GunPowder [41] is a test input generation tool for search-based test data generation using deep RL. GunPowder has been specifically developed for C applications and consists of three phases: (i) instrumentation, (ii) execution, and (iii) fitness evaluation. In the instrumentation phase, in the first step, it adds instrumentation codes that allow the tool to control and monitor the execution of the program. Subsequently, in the second step, the tool builds and executes the program, and in the third phase, the machine learning algorithm was applied to generate test inputs. Currently, the fitness function supported by GunPowder aims to improve branch coverage. Although not suitable for Android app testing, other studies have adopted RL techniques for Android testing [24,42,43].

Vuong and Takada [42] proposed a Q-Learning-based automated test case generation tool designed for Android apps using the Markov model to describe the AUT. The tool learns the most relevant behavioral model of the AUT and generates test cases based on this model. The tool executes a sequence of a fixed number of events, also called an episode. After finishing an episode, the tool selects a random state from those that have already been visited and starts a new episode in the next phase. However, this tool has multiple limitations. For example, it only generates UI events and does not cover activities triggered by system events.

Table 1. Overview of test case generation tools for Android apps.

No	Tool	Technique	Test Case Generation Approach	Test Inputs	Testing Environment	Test Artifacts	Basis	Availability
1	APE	Dynamic analysis	Model-based	User	Real Device, Emulator	Crash, coverage reports	Monkey	Yes
2	Humanoid	Dynamic analysis	Deep Q Network	User, System	Real Device, Emulator	Log and coverage report	Droidbot	Yes
3	AndroFrame	Dynamic analysis	Q-Learning-Based	User, System	Real Device, Emulator	Log	-	No
4	(Adamo, Khan, et al., 2018)	Dynamic analysis	Q-Learning-Based	User	Emulator	Coverage report	Appium	No
5	(Vuong and Takada, 2018)	Dynamic analysis	Q-Learning-Based	User	Real Device, Emulator	Log and coverage report	UI Automator	No
6	AimDroid	Dynamic analysis	Model-based/SARSA	User	Real Device, Emulator	Crash and coverage reports	Monkey	Yes
7	DroidBot	Dynamic analysis	Model-based	User, System	Real Device, Emulator	Log and coverage report	-	Yes
8	SmartMonkey	Dynamic analysis	Random based	User, System	-	-	Monkey	No
9	Stoat	Hybrid	Model-based	User, System	Real Device, Emulator	Log, crash and coverage reports	A3E	Yes
10	Sapienz	Hybrid	Search-based/Random	User, System	Emulator	log, crash and coverage report	Monkey	Yes
11	Crashscope	Hybrid	Systematic	User, System	Real Device, Emulator	Crash report	-	No
12	Sig-Droid	Static analysis	Systematic	User	Emulator	Log	Java PathFinder	Yes
13	AppDoctore	Hybrid	Random based	User	Real Device	Log	Monkey	Yes

Table 1. *Cont.*

No	Tool	Technique	Test Case Generation Approach	Test Inputs	Testing Environment	Test Artifacts	Basis	Availability
14	DroidCrawle	Dynamic analysis	Model-based	User	Emulator	Coverage report	-	No
15	Puma	Dynamic analysis	Model-based	User	Real Device, Emulator	Log	Monkey	Yes
16	Dynodroid	Dynamic analysis	Guided/Random	User, System	Emulator	Coverage and crash reports	Monkey Runner	Yes
17	ORBIT	Dynamic analysis	Model-based	User	Real Device, Emulator	-	-	No
18	A3E-Targeted	Static analysis	Systematic	User, System	Real Device, Emulator	-	Troyd	No
19	(Hu and Neamtiu, 2011)	Dynamic analysis	Random	User	Emulator	Log	Monkey	No
20	Monkey	Dynamic analysis	Random based	User, System	Real Device, Emulator	Log	-	Yes

Adamo, Khan, Koppula, and Bryce [43] introduced a Q-Learning-based automated test case generation tool designed for Android apps built on the top of Appium and UI Automator. During the process of test case generation, the tool chooses an event with the highest Q-value from the set of available events in each state. The test case generation process is quite similar to previous work [42], and this generation process is divided into episodes where the states used in previous episodes are employed as a basis for beginning a new episode. The authors define the state to be the set containing the unique actions available.

AndroFrame [24] is a Q-Learning-based exploration tool that generates test cases. Instead of using a random based approach, the GUI is explored based on a pre-approximated probability distribution that satisfied a test objective. It creates a Q-matrix that shows the probabilities of reaching the test objective which is used to select the next action. However, AndroFrame has inconsistent activity coverage and only works with single-objective fitness functions, where each run has only one objective to increase the activity coverage or search crashes.

5.2. Automated GUI Testing with Reinforcement Learning

AimDroid [44] is a model-based test case generation tool for Android apps. AimDroid implements an RL-guided random approach. AimDroid is composed of two activities: it runs a breadth-first search to discover unexplored activities and insulates the discovered activity in a "cage" and intensively exploits such activity using RL-guided fuzzing algorithms. This tool divides the tests into episodes; each episode generates a bounded number of events and focuses on a single activity by disabling activity transitions. Furthermore, AimDroid uses an RL algorithm called SARSA to learn about the ability of events that can discover new activities, to "look ahead" and to select events that are more likely to trigger new activities and crash greedily. AimDroid also has some limitations. For example, it disables the activity transition, which may drop some faults caused by the activity life cycle. Moreover, AimDroid does not learn the second-best event to choose from, it only knows the best SARSA-based event, and for all other events, it chooses randomly.

Humanoid [29] was developed along with DroidBot [28] which was introduced to learn how users interact with Android apps. Humanoid uses a GUI model to comprehend and analyze the behavior of AUT. Nevertheless, Humanoid gives preference to human-interacted UI components. Humanoid works in two stages; (1) offline learning phase which is a deep neural network model used to master the relationship between GUI contexts and user-performed interactions, and (2) online testing phase where Humanoid developed a UI transition model for the AUT. In the second phase, it uses the UI transition model and the interaction model to determine the type of test input to send. The UI transition model guides Humanoid on the navigation of explored UI states, while the interaction model guides the discovery of the new UI states. As a limitation, this tool does not present an increment in coverage when compared to other tools. It is unable to use textual information available in the app to generate test cases.

5.3. Automated GUI Testing Approaches

Several approaches have been developed to facilitate test case construction, and to explore Android apps: (i) random-based, (ii) model-based, and (iii) systematic testing.

Random-based testing is one of the popular techniques to detect system-level faults within the app. It floods an app with unfeasible events that fail to explore all the app's functionalities [45]. This approach is used to generate events that make them suitable for stress testing efficiently. Android Monkey [27] is a black-box GUI testing tool in the Android SDK (Software Development Kit). Among the current test generation tools, this random-based testing tool has gained considerable popularity from society. Apart from its simplicity, it has demonstrated good compatibility with a myriad of Android platforms, making it the most commonly used tool for numerous industrial applications [19,46]. However, Android Monkey requires more time to generate a long sequence of events. These events include

redundant events that repeatedly jump between app activities and unfruitful events that click on a non-interactive area on the screen [44,47,48]. Hu et al. [49] developed an approach on top of Android Monkey to automatically detect UI crashes. Dynodroid [13] used two heuristics algorithms and these algorithms select relevant events to the app's current state and repeat the process in the observe-select-execute cycle. However, Dynodroid has used instrumentation to infer relevant events to guide exploration. In contrast, UI-guided inputs without instrumentation were generated from our approach. SmartMonkey [50] uses a random testing approach proposed by [45] to reduce the number of test cases and the time required to identify the first fault. It constructs a transition model of the app based on random interaction and generates test cases through the random walk technique. SmartMonkey generates test cases consist of a sequence of user and system events.

Model-based testing uses a graph-based model to represent the user interaction with the app's GUI. The model is designed either manually or automatically by adopting the AUT's specifications, such as code or XML configuration files, or through direct interaction with the apps. Model-based exploration can be guided to specific unexplored parts using a systemic strategy such as depth-first exploration, breadth-first exploration, or hybrid [23], or a stochastic model [15]. The model-based technique has encountered difficulties in inaccurate modeling. More specifically, dynamic behaviors in GUIs can generate inaccurate model or state explosion issues due to non-deterministic changes in GUIs. Hence, the model-based approach ignores the changes, finds the event unimportant, and then proceeds with the discovery differently. Explicitly, a GUI model that includes only a limited range of possible behavioral spaces will minimize the effectiveness of tests. Baek et al. [51] conducted a multi-level state representation study to show that different levels of abstraction have an impact on the effectiveness of the modeling tool. A3E [23] explores apps with two strategies: depth-first exploration, which systematically analyzes apps while running on the actual devices and without access to the source code, and targeted exploration, which prioritizes exploration of activities that begin from the initial activity on a static activity transition graph. A3E represents every activity as an individual state without considering that the activity can exist in different states. This leads to a lack of some behaviors of the application as not all states of the activities are being explored. Orbit [52] statically analyzes the apps' source code to generate relevant events supported by the app. However, it uses a simple depth-first exploration that restarts the app from its initial state to backtrack to previous states. PUMA [53] includes a generic UI Automator that implements the same basic random approach as Android Monkey; however, it differs in its design and uses dynamic analysis to trigger changes in the environment during the app execution. Stoat [15], performs a stochastic model testing in two phases. First, it creates a probabilistic model by exploring and analyzing the apps GUI interactions dynamically. Second, it optimizes the state-model by performing Gibbs sampling and directs test generation from the optimized model to maximize code and activity coverage. Ape [54] uses a dynamic modeling approach to optimize the initial GUI model by leveraging runtime information.

Systematic testing uses more sophisticated techniques, such as symbolic execution and evolutionary algorithms to generate specific inputs. The strength of this approach is that it can leverage the source code to generate tests to reveal previously uncovered application behavior. Thor [55] executes the existing test cases under adverse conditions. However, Thor does not generate test cases and relies on injecting existing test cases with sequences of events, which do not affect the outcome of the original test cases. CrashScope [56] automatically detects AUT crashes using a hybrid strategy that combines systematic exploration and static analysis. AppDoctor [57] uses an approximate execution approach to speed up testing and automatically classify most reports into bugs or false positives. However, this approach offers the ability to replay bugs and introduces stack traces to the developer. It must replay crash traces to prune false-positives, and it does not offer highly detailed and expressive reports. SIG-Droid [58] is an automated system input generation tool for Android apps that combines program analysis techniques with symbolic execution to achieve high code coverage. Sapienz [16] uses a fully automated multi-objective search-

based testing approach. It adapts genetic algorithms to optimize the test sequence to maximize code coverage and fault detection while minimizing the test sequence length. Sapienz explores the app components by using the specific GUIs and complex sequences of input events with a pre-defined pattern. This pre-defined pattern is termed the motif gene that capture the experience of the testers. Thus, it produces a higher code coverage by concatenating the atomic events.

6. Proposed Approach: Q-Learning to Generate Test Case for Android Apps

The idea behind using Q-Learning is that the tabular Q-function is rewarded with each selection of possible actions over the app. However, this reward may vary according to the test objective. Thus, events that are never selected can present a higher reward than events that have already been executed, which reduces the redundant execution of events and increases coverage.

Q-Learning has been used in software testing in the past and has shown better results to improve the random exploration strategy [24,42,43,59]. However, a common limitation to all these tools is that the reward function assigns the highest reward when the event is executed for the first time to maximize coverage or locate crashes. Nonetheless, in the proposed approach, the environment does not offer direct rewards to the agent. The agent itself tries to visit all states to collect more rewards. The proposed approach uses tabular Q-Learning like other approaches but uses an effective exploration strategy that reduces actions redundant execution and uses different states and action spaces. Action selection is the main part of Q-Learning in finding an optimal policy. The policy is a process that decides on the next action a from the set of current actions. Unlike previous studies, the proposed approach utilizes the upper confidence bound (UCB) exploration-exploitation strategy as a learning policy to create an efficient exploration strategy for GUI testing. UCB tries to ensure that each action is explored well and is the most widely used solution for multi-armed bandit problems [60]. The UCB strategy is based on the principle of optimism in the face of uncertainty.

6.1. Implementation

Q-Learning technique with UCB exploration strategy was adopted to generate a GUI test case for Android apps to improve coverage and crash detection. This approach was built in a test tool named DroidbotX. Moreover, the main idea of using DroidbotX was to evaluate the practical usefulness and applicability of the proposed approach. DroidbotX works with Droidbot [28]. Droidbot is a UI-guided input generation tool used mainly for malware detection and compatibility testing. Droidbot was chosen because it is open-source and can test apps without having access to the apps' source code. Moreover, it can be used on an emulator or real device without instrumentation and is compatible with all Android APIs. The DroidbotX algorithm tries to visit all states because it assumes "optimism in the face of uncertainty". The principle of optimism in the face of uncertainty is known as a heuristic in sequential decision-making problems, which is a common point in exploration methods. The agent believes that it can obtain more rewards by reaching the unexplored parts of the state's space [61]. In this principle, actions are selected greedily, but strong optimistic prior beliefs are put on their payoffs so that strong contrary evidence is needed to eliminate the action from consideration. This technique has been used in several RL algorithms, including the interval exploration method [62]. In other words, it means that visiting new states and making new actions would bring the agent more reward than visiting old states and making old actions. Therefore, it starts from an empty Q-function matrix and assumes that every state and action reward an agent with $+1$. When it visits the state s and makes an action a, the Q-function Q (s, a) decreases, and the priority of the action a for the state s becomes lower. Our DroidbotX approach generates sequences of test inputs for Android apps that do not have an existing GUI model. The overall DroidbotX architecture is shown in Figure 2.

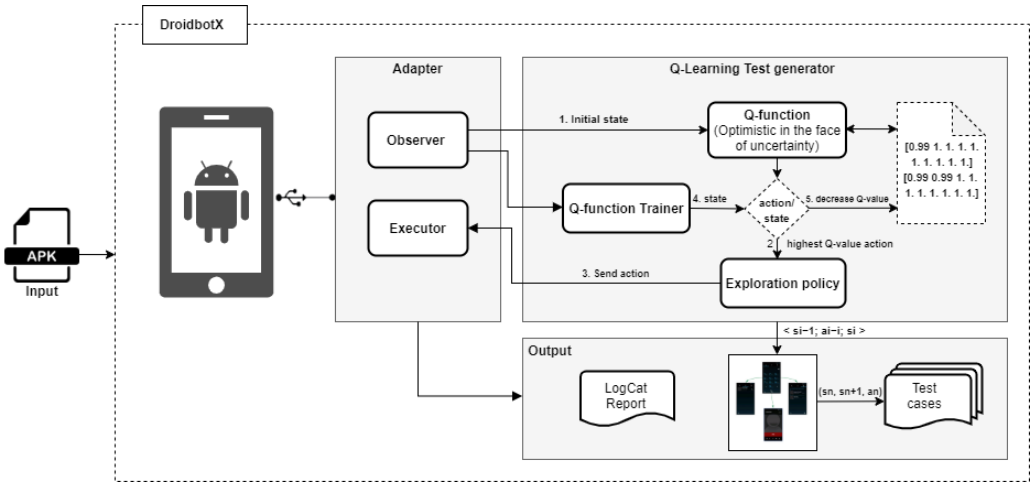

Figure 2. Overview of DroidbotX.

In Figure 2, the adapter acts as a bridge between the test environment and the test generation algorithm. The adapter is connected to an Android device or an emulator via the Android Debug Bridge (ADB). The adapter observer monitors the AUT and sends the current state to the test generator. Simultaneously, the executor receives the test inputs generated by the algorithm and translates them to commands. Furthermore, the test generator interacts and explores the app's functionalities following the observe-select-execute strategy, where all the GUI actions of the current state of AUT are observed; one action is selected based on the selection strategy under consideration, and the selected action is executed on the AUT. Similar to other test generators, DroidbotX uses a GUI model to save the memory of transitions called a UI transition graph (UTG). The UTG guides the tool to navigate between the explored UI states. The UTG is dynamically constructed at runtime, which is a directed graph whose nodes are UI states, and the edges between the two nodes are actions that lead to UI state transitions. The state node contains the GUI information and the running process information, and the methods are triggered by the action. DroidbotX uses Q-Learning-based test coverage approach shown in Algorithm 1 and constructs UI transition graph in Algorithm 2.

6.2. States and Actions Representation

In the Android app, all the UI widgets of an app activity are organized in a GUI view tree [51]. The GUI tree can be extracted via UI Automator, which is a tool provided by the Android SDK. UI widgets include buttons, text boxes, search bars, switches, and number pickers. Users interact with the app using click, long-click, scroll, swipe up, swipe down, input text, and other gestures collectively called as GUI actions or actions. Every action is represented by its action type and target location coordinates. The GUI action is either (1) widget-dependent such as click and text, or (2) widget-independent such as the back that presses the hardware back button. A 5-tuple denotes an action: $a = (w, t, v, k, i)$, where w is a widget on a particular state, t is a type of action that can be performed on the widget (e.g., click, scroll, swipe), and v holds arbitrary text if widget w is a text field. For all non-text field widgets, the v value is empty. Moreover, k is the key event that includes back, menu, and home buttons on the device, and i is a widget ID. Note that DroidbotX sends an intent action that installs, uninstalls, and restarts the app.

State abstraction refers to the procedure that identifies equivalent states. In this approach, state abstraction determines two states as equivalent if (1) they have similar GUI content which includes package, activity, widget's type, position, and widgets parent-child relationship, and (2) they have the same set of actions on all interactive widgets, which is widely used in previous GUI testing techniques [44,63,64]. GUI state or state $s \in S$ describes the attributes of the current screen out of the Android device where S denotes the set of all states. A content-based comparison and a set of actions to decide state equivalence,

where two states with different UI contents and different enabled actions are assumed to be different states.

For simplifying states and actions representation, take an example of the Hot death app. Hot death is a variation of the classic card game. The main page includes a new game, settings, help, about, and exit buttons. Figure 3 shows a screenshot of the app's main activity, initial state, and related widgets with a set of enabled actions. Widget-dependent action is detected when a related widget exists on the screen. For example, a click-action exists only if there is a related widget with the attribute clickable true. Widget-independent action is available in all states because the user can press on device hardware buttons such as the home all the time.

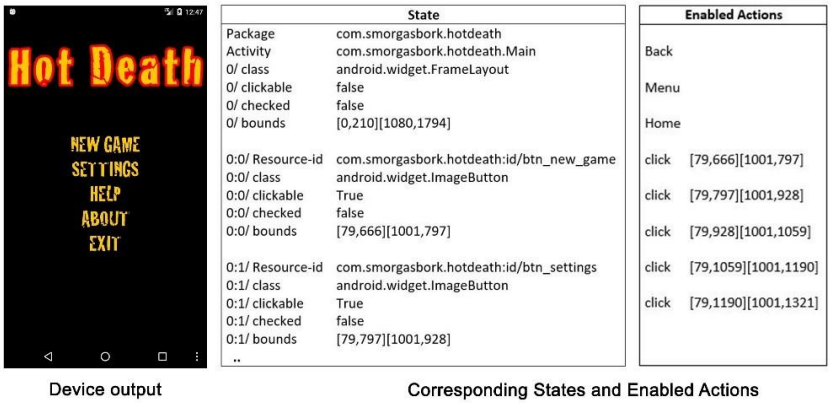

Figure 3. Displays state and actions representation from Android app.

6.3. Exploration Strategy

Android apps can have complex interactions between the events that can be triggered by UI widgets, and states that can be reached, and the resulting coverage achieved. In automated testing, the test generator must choose not only which widget to interact with, but also what type of action to perform. Each type of action on each widget is likely to improve coverage. Our goal is to interact with the app's widgets by sending relevant actions for each widget dynamically. This reduces the number of ineffective actions performed and explores as much app state as possible. Thus, UCB was used as an exploration policy to explore the app for new states and try out new actions. For each state, all potential widgets are extracted with their IDs and location coordinates, and then systematically choose between five different actions (i.e., click, long-click, scroll, swipe left/right/up/down, and input text data) to interact with each widget. Next, whether the action brings the app to a new state by comparing its contents with all other states in the state model. If the agent identifies a new state, the exploring policy on the new state is recursively applied to discover unexplored actions. The exploration policy does not know about the consequences of each action, and the decision is made based on the Q-function.When exploration of this state terminates, intent was executed to restart the AUT. Android intent is the message that passed between Android app components such as the start activity method to invoke activity. Examples of termination, an action that cause the AUT to crash, an action that switches to another app, or a clicks home button. The home action always closes the AUT, while the back action often closes the AUT. The exploration passes the login screen by searching in a set of pre-defined input. Some existing tools such as Android Monkey [27] will stop at the login screen, failing to exercise the app beyond the login page.

6.3.1. Observer and Rewarder

The goal of the observer is to monitor the results of actions on the AUT. The Q-function then rewards the actions based on the results. Algorithm 1 uses the input parameters to explore the GUI and produces a set of event sequences as a test case for AUT. The Q-function $Q(s, a)$ takes state s and action a. The Q-function matrix is constructed based on the current

state. Each row in the matrix represents the expected Q-values for a particular state. The row size is equal to the number of possible actions for the state. The *getEventfromActor* function at lines 23–26 obtains all the GUI actions of the current state of AUT. The actions' initial values on the current state are assigned as 1 at line 26. The *UpdateQFunction* function at lines 13–21 decreases the value of the action to 0.99 when the test generator conducts this action in the state. When all action value is 0.99, the maximum value becomes 0.99, and the test generator starts to choose some actions again. Then one action is selected and executed, and when a new state is found, the Q-function trainer receives the next state and updates the Q-function matrix to the previous state. The test generator sends *KeyEvents* such as back button at lines 27–28, if the state is the last or if there are no new actions in the current state.

$$Q(s,\mathrm{a}) = Q(s,\mathrm{a}) + \alpha * \left(\mathrm{r} + \gamma * maxQ\left(s',\mathrm{a}'\right) - Q(\mathrm{s},\mathrm{a})\right) \tag{2}$$

The Q-Learning algorithm uses Equation (2) to estimate the value of $Q(s,\mathrm{a})$ iteratively. The Q-function is initialized by a default value. Whenever an agent executes an action a from state s to reach s' and receives a reward $r\ +1$, the Q-function is updated as Equation (2) where α is a learning rate parameter between 0 and 1 and γ is a discount rate.

6.3.2. Action Selector

Action selection strategy is a crucial feature of DroidbotX. Right actions can improve the likelihood and decrease the time necessary to navigate to various app execution states. In the initial state, the test generator chooses the first action based on a randomized exploration policy to avoid the systematic handling of GUI layouts in each state. Then, the test generator selects actions from the new states and generates event sequences in a way that attempts to visit all states. The Q-function calculates the expected future rewards for actions based on the set of states it visited. In each state, the test generator chooses an action that has the highest expected Q-value from the set of available actions using *getSoftArgmaxAction* function at lines 32–36, then the predicted Q-value for that action is reduced. Therefore, the test generator will not choose it again until all other actions have been tried. Formalizing this mathematically, the selected action is picked by Equation (3).

$$action = \underset{a}{\operatorname{argmax}} \left[Q_t\left(s_t, a^i\right) + \sqrt[c]{\frac{IogN_{s_t}}{N\left(s_t, a^i\right)}} \right] \tag{3}$$

Equation (3) depicts the basic idea of UCB strategy, the expected overall reward of action a is $Q_t\left(s_t, a^i\right)$, $IogN_{s_t}$ denotes how often action has been selected in s_t, while $N\left(s_t, a^i\right)$ is the number of times the action a^i was selected in state s_t, and c is a confidence value that controls the level of exploration (set to 1). This method is known as "exploration through optimism," and it gives less-explored action a higher value and encourages the test generator to select them. The test generator uses the Q-function learned by Equation (2) and UCB strategy to select each action intelligently, which balances the exploration and the exploitation of AUT.

6.3.3. Test Case Generation

Test case TC is defined as a sequence of transitions. $TC = (s1,\ a1,\ s2),\ (s2,\ a2,\ s3),\ \ldots,\ (sn,\ an,\ sn+1)$, where n is the length of the test case. Each episode is considered to be a test case, and each test suite TS is a set of test cases. The transition is defined as a 3-tuple (start-state, *ss*; action, *a*; end-state, *se*). Algorithm 2 dynamically constructs a UI transition graph to navigate between the explored UI states. It takes three input parameters: (1) the app under test, (2) Q-function for all the state-action pairs generated by Algorithm 1, and (3) test suite completion criterion. The criterion for test suite completion is a fixed number of event sequences (set to 1000). DroidbotX's test generator explores a new state si, and adds a new edge $\langle si-1;\ ai-i;\ si\rangle$ to the UI transition graph, where $si-1$. is the last observed UI state and $ai-i$ is the action performed in $si-1$. For instance, consider

generation of a test suite for Hot death Android app. DroidbotX creates an empty UI transition graph *G* (line 1), explores the current state of AUT (line 3), observes all the GUI actions of the current state (line 5), and constructs a Q-function matrix. Then one action is selected and executed based on *getSoftArgmaxAction* function (line 7), when a new state is found, the *UpdateQFunction* function receives the next state and updates the Q-function matrix to the previous state. The transition of executed action, next state, and previous state are added to the graph (line 15). The Q-value of executed action is decreased to avoid using the same action of the current state. The process is repeated until completing the target number of actions. Figure 4 shows an example of UTG from the Hot death Android app.

Algorithm 1: Q-Learning based test generation

```
Input:   A, Application under test
Output:  S, set of states;
         Q, q-function for all the state-action pairs;
         P, transition matrix, epsilon;
         KeyEvent -exploration parameter
1  (S, Q, P) ← (Ø, Ø, Ø)
2  launch(A)
3
4  while true do
5          Event ← getEventfromActor(Q)
6          Update P[old_state, new_state, :] #adjusting P[old_state, new_state, Event]
7          Q← UpdateQFunction(Q, P)
8          Execute(Event)
9          If not enable:
10                 break
11 return (S, Q, P)
12
13 Function UpdateQFunction(Q, P)
14     Q_target ← (Ø)
15     for index in [0, 1, . . . , 9] do
16         for s in S do
17             Q_target[s] ← maximum of Q[s, event] for all events
18         for s in S do
19             for a in all events that was ever made do
20                 Q[s, a] ← 0.99 * sum(Q_target[:] * P[s, :, a])
21     return Q
22
23 Function getEventfromActor(Q)
24     state ← getCurrentState()
25     if state is not in S:
26         Q[state, :] ←1 # For all possible events from state
27     if RANDOM([0; 1]) < epsilon do
28         event ←KeyEvent
29     else
30         event ←getSoftArgmaxAction(Q[state])
31     return event
32 Function getSoftArgmaxAction(Q_state)
33     max_qvalue ← max(Q_state)
34     best_actions ← all events where Q_state[event] == max_qvalue
35     event ← choose randomly from best_actions
36     return event
```

Algorithm 2: DroidbotX Test Suite Generation Algorithm

Input: **AUT, App under test**
Input: **Q, Q-function for all the state-action pairs**
Input: **C, Test suite completion criterion**
Output: TS, Test Suite

Create an empty UI transition graph $G = \langle S, E \rangle$
Run the AUT
Observe current UI state s **and add** s **to** S
repeat
    Get All unexplored actions in s as A
    if A is not empty then
    Select as action a from A based on $Q(s)$
else
Extract a state s in S that has unexplored actions
    Get the shortest path p from s to s in G
    Select the first action in p as a
end if
Perform action a
Observe the new UI state s new and add s new to S
Add the edge $\langle s, a, snew \rangle$ to E
Until all actions in all states in S have been explored
or
Until length of TS is equal c
end

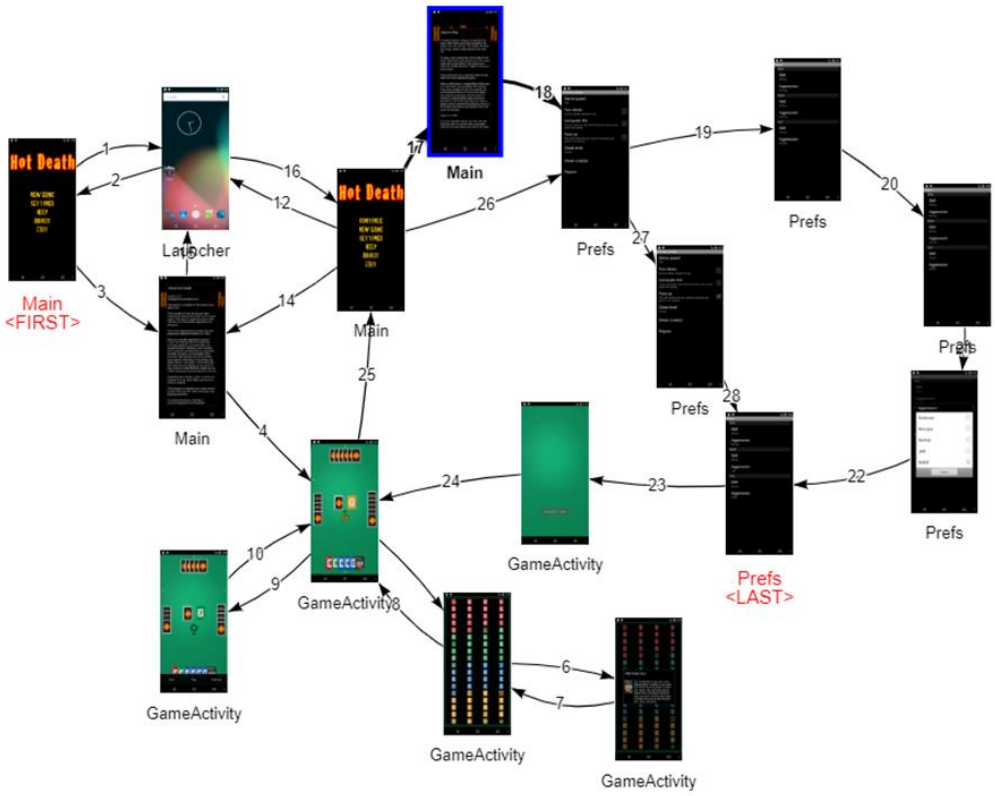

Figure 4. Shows a UI state transition graph from a real-world Android app (Hot death).

7. Empirical Evaluation

This section provides an evaluation of DroidbotX compared to state-of-the-art tools using 30 Android apps. This evaluation employs the empirical case study method that is

used in software engineering, as reported in [65,66]. The evaluation intends to answer the following research questions:

RQ.1: How does the coverage achieved by DroidbotX compared to the state-of-the-art tools?
RQ.2: How effective is DroidbotX to detect unique app crashes compared to the state-of-the-art tools?
RQ.3: How does DroidbotX compare to the state-of-the-art tools in terms of test sequence length?

7.1. Case Study Criteria

Four criteria were used for the evaluation as follows: (1) Instruction coverage refers to the Smali [67] code instructions through decompiling the APK installation package. It is the ratio of triggered instruction in the Java instruction code of the app to the total number of instructions. Huang et al. [68] first proposed the concept of instruction coverage, which is used in many studies as an indicator to evaluate test efficiency [24,44,54,64]. It is a more accurate and valid test coverage criterion that reflects the adequacy of testing results for closed-source apps [25]. (2) Method coverage is the ratio of the number of methods called during execution of the AUT to the total number of methods in the source code of the app. By improving the method coverage, more functionalities of the app were explored and tested [11,23,24,26]. (3) Activity coverage is defined as the ratio of activities explored during execution to the total number of activities existing in the app. (4) Crash detection: An Android app crashes when there is an unexpected exit caused by an unhandled exception [69]. Crashes will result in the termination of the app's processes, and dialogue is displayed to notify the user about the app crash. The further the code the tool explores, the more likely it is to discover potential crashes.

7.2. Subject Selections

We used 30 Android apps chosen from F-Droid repository [30] for the experimental analysis. These apps were chosen from the repository based on the app's number of activities and user permissions required. These features were determined in the Android manifest file of the app. User permissions were selected to evaluate how tools react to different system events such as call logs, Bluetooth, Wi-Fi, location, and the camera of the device. Table 2 lists the apps by app type, along with the package name, the number of activities, methods, and instructions in the app (which offers a rough estimate of the app size). Acvtool [70] was used to collect instruction coverage and method coverage. This tool does not require the source code of the app.

7.3. Experimental Setup

Our experiments were executed on a 64-Bit Octa-Core machine with a 3.50 Gigahertz Intel Xeon® central processing unit (CPU) running on Ubuntu 16.04 and 8 Gigabytes of RAM. Five state-of-the-art test generation tools for Android apps were installed on the dedicated machine for running our experiments. The tools chosen were Sapienz [16], Stoat [15], Droidbot [28], Humanoid [29], and Android Monkey [27].

The Android emulator x86 ABI (Application Binary Interface) image was used for experiments. All comparative experiments ran on emulators because the publicly available version of Sapienz only supports emulators. In contrast, DroidbotX, Droidbot, Humanoid, Stoat, and Android Monkey support both emulators and real devices. Moreover, Sapienz and Stoat ran on the same version of Android 4.4.2 (Android KitKat, API level 19) because of their compatibility as described in previous studies [15,16]; DroidbotX, Droidbot, Humanoid, and Android Monkey ran on Android 6.0.1 (Android Marshmallow, API level 23).

Table 2. Overview of Android apps selected for testing.

No	APP Name	Package Name	Version	Category	Instruction	Methods	Activity
1	Bubble	com.nkanaev.comics	4.1	Books	5208	463	2
2	WLAN Scanner	org.bitbatzen.wlanscanner	4	Communication	2484	141	1
3	Divide	com.khurana.apps.divideandconquer	2.1	Education	2306	195	2
4	Raele.concurseiro	raele.concurseiro	3	Education	1299	444	2
5	LolcatBuilder	com.android.lolcat	2.3	Entertainment	2497	79	1
6	MunchLife	info.bpace.munchlife	2.3	Entertainment	551	39	2
7	Currency	org.billthefarmer.currency	4	Finance	5461	148	5
8	Boogdroid	me.johnmh.boogdroid	4	Game	3984	398	3
9	Hot Death	com.smorgasbork.hotdeath	2.1	Game	17,679	365	3
10	Resdicegame	com.ridgelineapps.resdicegame	1.5	Game	6853	144	4
11	Pushup Buddy	org.example.pushupbuddy	1.6	Health and Fitness	1985	165	7
12	Mirrored	de.homac.Mirrored	2.3	Magazines	3803	219	4
13	A2DP Volume	a2dp.Vol	2.3	Maps and Navigation	13,452	600	8
14	Ethersynth	net.sf.ethersynth	2.1	Music and Audio	4056	168	8
15	Adsdroid	hu.vsza.adsdroid	2.3	Productivity	488	199	2
16	Talalarmo	trikita.talalarmo	4	Productivity	5122	658	3
17	Alarm Clock	com.angrydoughnuts.android.alarmclock	2.7	Productivity	5207	334	5
18	World Clock	ch.corten.aha.worldclock	2.3	Productivity	5200	315	4
19	Blockinger	org.blockinger.game	2.3	Puzzle	7090	356	6
20	Applications info	com.majeur.applicationsinfo	4.1	Tool	4806	315	6
21	Dew Point	de.hoffmannsgimmickstaupunkt	2.1	Tools	2282	75	3
22	drhoffmann	de.drhoffmannsoftware	1.6	Tools	5171	164	9
23	List my Apps	de.onyxbits.listmyapps	2.3	Tools	1930	96	4
24	Sensors2Pd	org.sensors2.pd	2.3	Tools	1346	149	4
25	Terminal Emulator	jackpal.androidterm	1.6	Tools	16,098	994	8

Table 2. *Cont.*

No	APP Name	Package Name	Version	Category	Instruction	Methods	Activity
26	Alogcat	org.jtb.alogcat	2.3	Tools	2344	199	3
27	Android Token	uk.co.bitethebullet.android.token	2.2	Tools	4658	288	6
28	Battery Circle	ch.blinkenlights.battery	1.5	Tools	963	79	1
29	Sensor readout	de.onyxbits.sensorreadout	2.3	Tools	994	683	3
30	Weather notifications	ru.gelin.android.weather.notification	2.3	Weather	8927	667	7

To achieve a fair comparison, a new Android emulator was used for each run to avoid any potential side-effects that may occur between the tools and apps. All tools were used with their default configurations. According to previous studies [11,54], Sapienz and Android Monkey were set to 200 milliseconds delay for GUI state updates. All testing tools were provided for an hour to test each app, similar to other studies [11,16,44]. To compensate for the possible effect of randomness during testing, each test was repeated five times (with each test consisting of one testing tool and one applicable app being tested). The final coverage and the progressive coverage were recorded after each action. Subsequently, the average value of the five tests was calculated as the final result.

8. Results

In this section, the research questions were answered by measuring and comparing four aspects: (i) instruction coverage, (ii) method coverage, (ii) activity coverage, and (iv) the number of detected crashes achieved by each testing tool on selected apps in our experiments. Table 3 shows the results obtained from the six testing tools. The gray background cells in Table 3 indicate the maximum value achieved during the test. The percentage value is the rounded-up value obtained from the average of the five iterations of the tests performed on each AUT.

RQ.1: How does the coverage achieved by DroidbotX compare to the state-of-the-art tools?

(1) Instruction coverage: The overall comparison of instruction coverage achieved by testing tools on selected Android apps is shown in Table 3. On average, DroidbotX achieved 51.5% instruction coverage, which is the highest across the compared tools. It achieved the highest value on 9 of 30 apps (including four ties, i.e., where DroidbotX covered the same number of instructions as another tool) compared to other tools. Sapienz achieved 48.1%, followed by Android Monkey (46.8%), Humanoid (45.8%), Stoat (45%), and Droidbot (45%).

Figure 5 presents the boxplots, where x indicates the mean of the final instruction coverage results across target apps. The boxes provide the minimum, mean, and maximum coverage achieved by the tools. Better results from DroidbotX can be explained as it accurately identifies which parts of the app are inadequately explored. The DroidbotX approach is used to explore the UI components by checking all actions available in each state and avoiding the use of the explored action to maximize coverage. In comparison, Humanoid achieved a 45.8% average value and had the highest coverage on 4 out of 30 apps due to its ability to prioritize critical UI components. Humanoid chooses from 10 actions available in each state that are likely to interact with human users.

Android Monkey's coverage was close to Sapienz's coverage during a one-hour test. Sapienz uses Android Monkey to generate events and uses an optimized evolutionary algorithm to increase coverage. Stoat and Droidbot achieved lower coverage than the other four tools. First, Droidbot explores UIs in depth-first order. Although this greedy strategy can reach deep UI pages at the beginning, it may get stuck because the order of the event execution is fixed at runtime. Second, Droidbot does not explicitly revisit the previously explored states, and this may fail to include a new code that should be reached by different sequences.

Table 3. Results on instruction coverage, method coverage, and activity coverage by test generation tools.

Apps Under Test	Instruction Coverage (%)						Method Coverage (%)						Activity Coverage (%)					
	M	**Sa**	**St**	**Dr**	**Hu**	**DrX**	**M**	**Sa**	**St**	**Dr**	**Hu**	**DrX**	**M**	**Sa**	**St**	**Dr**	**Hu**	**DrX**
Bubble	30.3	28.2	29.8	28.0	25.4	27.9	49.0	42.8	44.5	33.0	28.0	29.1	50.0	50.0	50.0	50.0	50.0	50.0
WLAN Scanner	58.9	61.3	57.2	59.2	58.6	59.4	63.5	66.0	59.3	65.4	64.4	65.5	100	100	100	100	100	100
Divide	60.8	57.4	55.4	56.5	57.4	59.1	71.7	52.8	47.2	54.5	54.5	63.9	100	100	100	100	100	100
Raele concurseiro	35.5	35.4	39.1	34.3	34.2	34.8	41.6	41.9	42.3	36.6	36.6	36.6	100	100	100	100	100	100
LolcatBuilder	23.2	23.2	21.6	24.5	24.6	24.9	29.9	32.9	27.8	32.9	32.9	32.9	100	100	100	100	100	100
MunchLife	73.6	75.0	75.7	72.8	76.5	75.1	62.1	66.7	66.7	66.2	66.7	66.2	100	100	100	100	100	100
Currency	47.0	49.1	50.3	38.6	45.4	49.0	55.8	58.8	61.5	40.4	53.8	58.1	88.0	100	100	92.0	96.0	100
Boogdroid	29.8	29.8	29.7	34.4	34.5	34.5	14.6	13.0	12.4	14.1	15.5	16.0	46.7	100	40.0	66.7	80.0	86.7
Hot Death	51.4	54.2	49.6	49.1	49.1	53.7	71.9	72.8	63.4	66.7	66.4	72.5	73.3	100	100	100	100	100
Resdicegame	67.2	71.5	64.9	66.2	61.6	72.9	52.6	62.4	51.3	60.6	53.6	66.8	100	100	100	100	100	100
Pushup Buddy	20.2	25.3	21.9	27.0	28.5	33.0	34.9	51.0	49.7	52.2	53.8	57.1	62.9	62.9	62.9	68.6	71.4	71.4
Mirrored	27.5	27.5	25.4	36.1	36.0	36.1	40.5	44.1	39.7	48.5	47.0	47.1	85.0	75.0	75.0	75.0	75.0	75.0
A2DP Volume	25.6	29.5	31.2	35.6	26.9	39.1	37.2	58.9	59.2	60.7	57.8	66.9	97.5	100	100	90.0	85.0	95.0
Ethersynth	71.5	77.4	55.7	47.9	64.8	82.7	68.8	78.5	67.1	61.8	68.1	85.2	92.5	100	100	100	100	100
Adsdroid	30.8	30.8	29.6	23.0	28.7	34.1	72.9	72.9	51.6	52.2	63.9	73.4	100	100	100	100	100	100
Applications info	29.4	45.7	38.9	57.7	45.7	68.3	40.5	58.8	51.8	61.6	53.8	65.1	100	100	100	100	100	100
Blockinger	66.0	66.4	66.5	67.0	67.9	69.3	62.5	77.3	78.5	79.1	79.2	82.0	100	100	100	100	100	100
Dew Point	68.2	71.9	68.9	67.0	72.9	72.8	61.9	75.7	73.9	74.9	75.7	75.7	100	100	100	100	100	100
drhoffmann	36.7	36.7	28.2	24.8	32.9	36.7	58.0	58.0	56.0	48.8	57.0	58.0	86.7	93.3	95.6	91.1	93.3	97.8
List my Apps	64.2	64.3	60.0	44.5	60.9	58.7	72.7	76.0	72.8	48.1	71.9	69.4	100	100	100	55.0	100	100
Sensors2Pd	74.1	73.6	71.3	71.1	74.2	70.4	86.6	81.9	83.2	81.6	90.6	80.7	100	100	100	100	100	100
Talalarmo	76.0	74.1	69.3	64.8	74.9	74.4	61.3	56.3	51.8	48.4	58.0	57.4	100	100	100	100	100	100
Terminal Emulator	40.0	41.4	34.1	35.6	36.2	40.9	50.7	54.4	46.2	52.2	51.2	53.9	35.0	37.5	37.5	37.5	37.5	37.5
Alarm Clock	59.6	49.1	48.5	66.3	61.8	64.1	52.8	49.7	50.2	81.0	66.8	79.9	76.0	60.0	92.0	96.0	64.0	100
Alogcat	56.7	62.7	74.8	50.6	51.1	70.1	69.4	78.9	92.2	73.2	75.5	80.9	66.7	66.7	66.7	66.7	66.7	66.7
Android Token	38.0	40.6	37.6	37.6	45.1	44.2	51.7	55.3	53.9	53.9	58.8	58.1	66.7	66.7	50.0	53.3	60.0	66.7
Battery Circle	77.6	78.1	78.1	74.0	81.3	81.3	83.5	84.1	83.5	76.5	85.6	83.5	100	100	100	100	100	100
Sensor readout	79.7	81.5	60.8	79.1	79.9	80.1	29.2	30.1	28.7	30.0	30.0	30.0	66.7	66.7	66.7	66.7	66.7	66.7
World Clock	44.4	41.3	42.5	24.8	33.8	46.1	54.5	43.8	54.9	39.5	39.1	57.5	70.0	95.0	95.0	85.0	95.0	100
Weather notifications	45.8	42.4	41.8	37.6	37.9	44.2	69.3	53.6	48.3	40.2	40.1	66.6	37.1	45.7	42.9	57.1	57.1	57.1
Overall	46.8	48.1	45.0	45.0	45.8	51.5	52.1	53.7	50.9	50.6	51.2	57.0	80.0	84.0	83.0	82.1	83.3	86.5

Keywords: M: Android Monkey, Sa: Sapienz, St: Stoat, Dr: Droidbot, Hu: Humanoid, DrX: DroidbotX

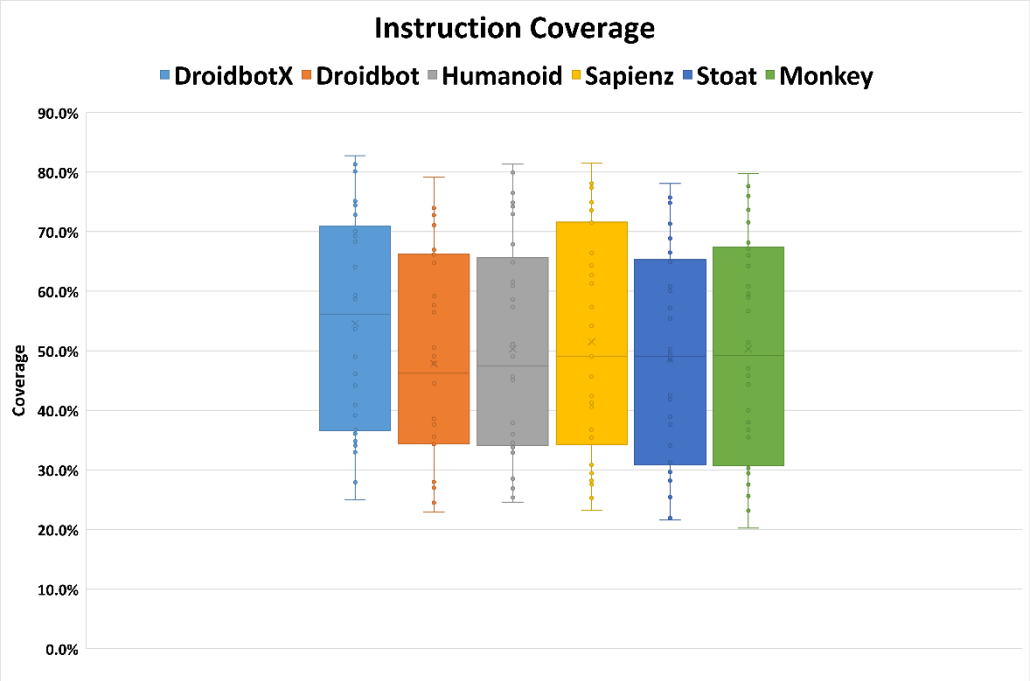

Figure 5. Variance of instruction coverage achieved across apps and five runs.

(2) Method coverage: DroidbotX significantly outperformed state-of-the-art tools in method coverage with an average value of 57%. The highest value was achieved on 9 out of 30 apps (including three ties where the tool covered the same method coverage as another tool). Table 3 shows that the coverage of app instructions obtained by the tools is lower than that of the method. This indicates that the method coverage cannot fully cover all the statements in the app's method. On average, Sapienz, Android Monkey, Humanoid, Stoat, and Droidbot achieved 53.7%, 52.1%, 51.2%, 50.9%, and 50.6% of method coverage, respectively. Stoat and Droidbot did not obtain the highest coverage of 50% on 10 of 30 apps after five rounds of testing. In contrast, DroidbotX achieved the highest method coverage of 50% in the 24 apps that were tested. In comparison, Android Monkey obtained less than 50% method coverage in eight apps. This study concluded that the AUT functionalities can be accomplished and explored using the observe-select-execute strategy and tested on standard equipment. Sapienz displayed the best method coverage on 5 out of 30 apps (including four ties where the tool covered the same method coverage as another tool). Sapienz's coverage was significantly higher for some apps such as "WLAN Scanner", "HotDeath", "ListMyApps", "SensorReadout", and "Terminal emulator". These apps have functionality that requires complex interactions with validated text input fields. Sapienz uses the Android Monkey input generation, which continuously generates events without waiting for the effect of the previous event. Moreover, Sapienz and Android Monkey can generate several events, broadcasts, and text that have not been supported by other tools. DroidbotX obtained the best results for several other apps, especially "A2DPVolume", "Blockinger", "Ethersynth", "Resdicegame", "Weather Notification", and "World Clock". The DroidbotX approach assigns Q-values to encourage the execution of actions that lead to new or partially explored states. This enables the approach to repeatedly execute high-value action sequences and revisit the subset of GUI states that provides access to most of the AUT's functionality.

Figure 6 presents the boxplots, where x indicates the mean of the final method coverage results across target apps. DroidbotX had the best performance compared to state-of-the-art

tools, and Android Monkey was used as a reference for evaluation in most Android testing tools. Android Monkey can be considered a baseline because it comes with an Android SDK and is popular among developers. Android Monkey obtained a lower coverage compared to DroidbotX because of its redundancy and random exploratory approach.

(3) Activity coverage: the activity coverage is measured by intermittent observation of the activity stack on the AUT and recording all activities listed down in the android manifest file. The activity coverage metric was chosen because, once DroidbotX has reached an activity, it can explore most of the activity's actions. The results determine activity coverage differences between DroidbotX and other state-of-the-art tools. The resulting average value of the tools revealed that the activity coverage performed better than instruction and method coverage, as shown in Table 3.

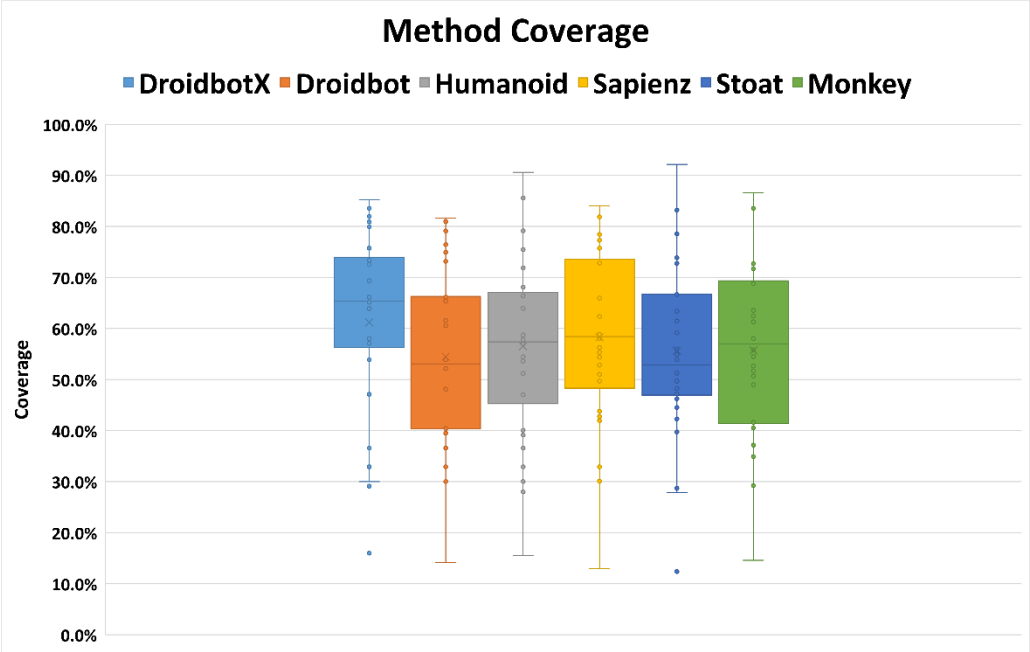

Figure 6. Variance of method coverage achieved across apps and five runs.

DroidbotX outperformed the other tools in its activity coverage, such as instruction and method coverage. DroidbotX has an average coverage of 86.5%, which was best achieved by the "Alarm Clock" app (including 28 ties, i.e., whereby DroidbotX covered the same number of activities as another tool). DroidbotX outperformed other tools because it did not explicitly revisit previously explored states due to its reward function. This was followed by Sapienz and Humanoid, with the average mean value of activity coverage at 84% and 83.3%, respectively. Stoat successfully outperformed Android Monkey in activity coverage with an average activity coverage of 83% due to an intrusive null intent fuzzing that can start an activity with empty intents. All tools under study were able to cover more than 50% of coverage on 25 apps, and four testing tools covered 100% activity coverage on 15 apps. Android Monkey, however, achieved less than 50% activity coverage of about three apps. Android Monkey achieved the least activity coverage with an average mean value of 80%.

Figure 7 shows the variance of the mean activity coverage of 5 runs across all 30 apps of the tool. The horizontal axis shows the tools used for the comparison. The vertical axis shows the percentage of activity coverage. Activity coverage was higher than the instruction and method coverage. DroidbotX, Droidbot, Humanoid, Sapienz, Stoat, and Android Monkey obtained a 100% coverage increased from a mean coverage of 89%, 85%, 86.6%, 87.3%, 85.8%, and 83.4%, respectively. All tools were able to cover above 50% of the activity coverage. Although Android Monkey implemented more types of events than

other tools, it achieved the least activity coverage. Android Monkey generates random events at random positions in the App activities. Therefore, its activity coverage can differ significantly from app to app and may be affected by the number of events sequences generated. To sum up, the high coverage of DroidbotX was mainly due to the ability of DroidbotX to perform a meaningful sequence of actions that could drive the app into new activities.

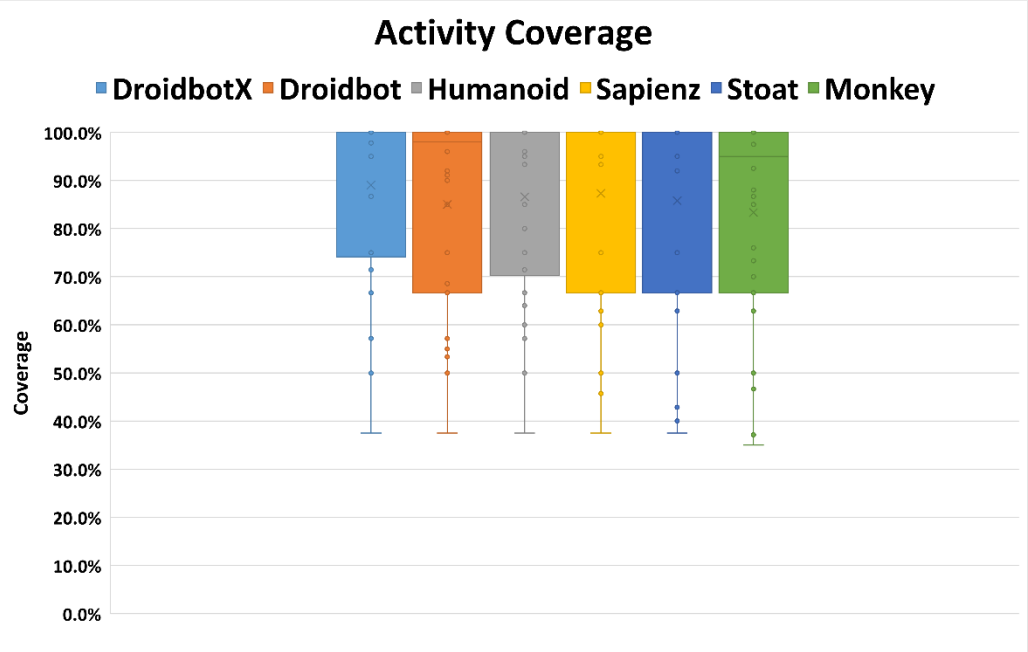

Figure 7. Variance of activity coverage achieved across apps and five runs.

RQ.2: How effective is DroidbotX in detecting unique app crashes compared to the state-of-the-art tools?

A crash is uniquely identified by the error message and the crashing activity. LogCat [71] is used to repeatedly check the crashes encountered during the AUT execution. LogCat is a tool that uses the command-line interface to dump logs of all the system-level messages. Log reports were manually analyzed to identify unique crashes from the error stack following the Su et al. [15] protocol. First, crashes unrelated to the app's execution by retaining only exceptions containing the app's package name and filter crashes of the tool itself, or initialization errors of the apps in the Android emulator. Second, compute a hash over the sanitized stack trace of the crash to identify unique crashes. Different crashes should have a different stack trace and thus a different hash. Each unique crash exception is recorded per tool, and the execution process is repeated five times to prevent randomness in the results. The number of unique app crashes is used as a measure of the performance of the crash detection tool. Crashes detected by tools on a different version of Android via normalized stack traces were not compared because different versions of Android have different framework code. In particular, Android 6.0 uses the ART runtime while Android 4.4 uses Dalvik VM, different runtime environments have different thread entry methods. Based on Figure 8, each of the tools compared complements the others in crash detection and has its advantages. DroidbotX triggered an average of 18 unique crashes in 14 apps, followed by Sapienz (16), Stoat (14), Droidbot (12), Humanoid (12), and Android Monkey (11).

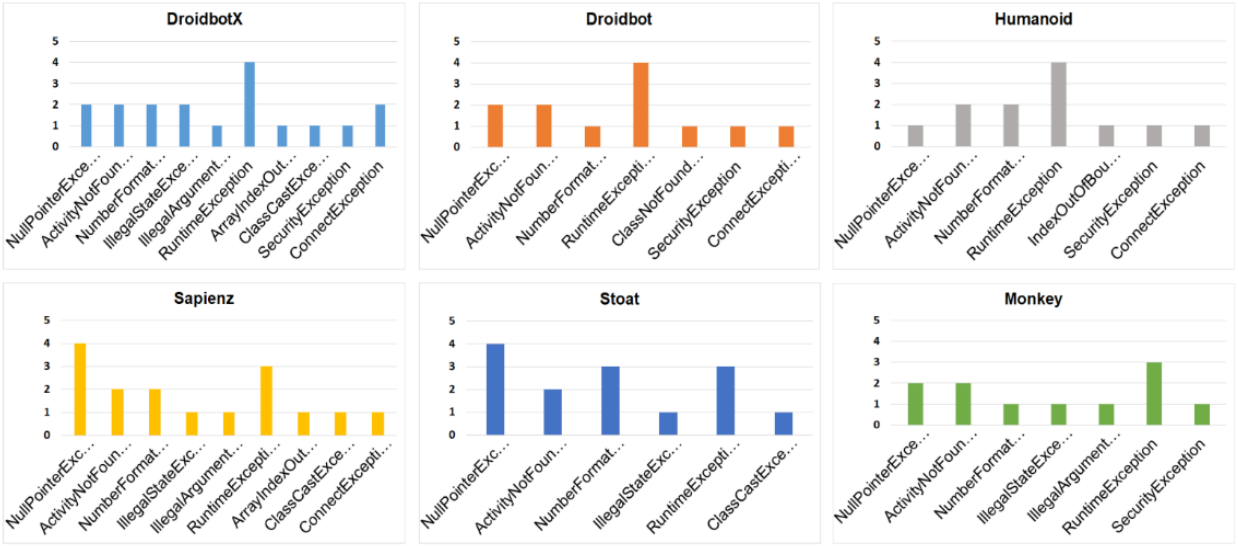

Figure 8. Distribution of crashes discovered.

Like activity coverage, Android Monkey remains the same as it has the least capacity to detect crashes due to its exploratory approach that generates a lot of ineffective and redundant events. Figure 8 summarizes the distribution of crashes by the six testing tools. Most of the bugs are caused by accessing null references. Common reasons are that developers forget to initialize references, access references that have been cleaned up, skip checks of null references, and fail to verify certain assumptions about the environments [57]. DroidbotX is the only tool to detect IllegalArgumentException on the "World Clock" app, because it is capable of managing the exploration of states, and systematically sends back button events that may change the activity life cycle. This bug is caused by an incorrect redefinition of the onPause method of activity. Android apps may have incorrect behavior due to mismanagement of the activity's lifecycle. Sapienz uses Android Monkey to generate an initial population of event sequences (including both user and system events) prior to genetic optimization. This allows Sapienz to trigger other types of exception, including ArrayIndexOutOfBoundsException, and ClassCastException. For the "Alarm Clock" app, DroidbotX and Droidbot detected a crash on an activity that was not discovered by other tools in the five runs. Manually inspected several randomly selected crashes to confirm that they do appear in the original APK as well, and found no discrepancy between the original and the instrumented APK behaviors.

RQ.3: How does DroidbotX compare to the state-of-the-art tools in terms of test sequence length?

The effectiveness of events sequence length on test coverage and crash detection was investigated. The event sequence length generally shows the number of steps required by the test input generation tools to detect a crash. It is critical to highlight its effectiveness due to its significant effects on time, testing effort, and computational costs.

Figure 9 depicts the progressive coverage of each tool over the threshold time used (i.e., 60 min). The progressive average coverage for all 30 apps was calculated every 10 min for each of the test generation tools in the study and a direct comparison of the final coverage was published. In the first 10 min, the coverage for all testing tools increased rapidly, as the apps had just started. At 30 min, DroidbotX achieved the highest coverage value compared to other tools. The reason is that the UCB exploration strategy implemented in DroidbotX finds events based on their reward and Q-value, which eventually tries to select and execute the previously unexecuted or less executed events, thus aiming for high coverage.

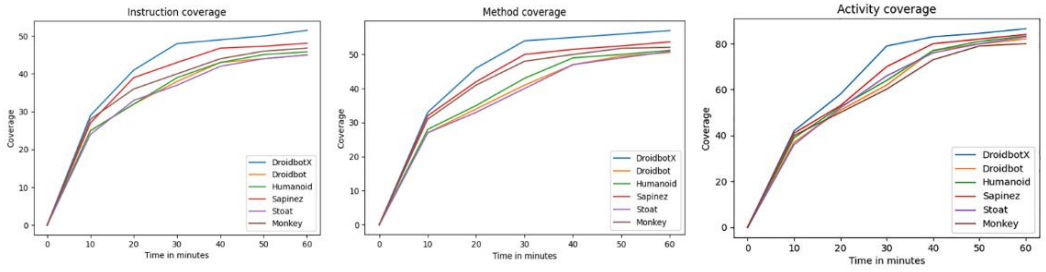

Figure 9. Progressive coverage.

Sapienz coverage increased rapidly, as the apps had just started, whereas all UI states were new but could not exceed the peak reached after 40 min. Sapienz has a high tendency to explore visited states, which could generate more event sequences. Stoat, Droidbot, and Humanoid had almost the same result and had better activity coverage than Android Monkey. Android Monkey could not exceed the peak reached after 50 min. The reason is that a random approach generates the same set of redundant events leading to a fall in its activity exploration ability. It is essential to highlight that these redundant events produced insignificant coverage improvement as the time budget increased.

Table 4 shows that the Q-Learning approach implemented in DroidbotX achieved 51.5% instruction coverage, 57% method coverage, 86.5% activity coverage, and triggered 18 crashes within the shortest event sequence length compared to other tools.

Table 4. Experimental results to answer research questions.

Tools	Instruction Coverage (%)	Method Coverage (%)	Activity Coverage (%)	Number of Crashes	Max Events Number
DroidbotX	51.5	57	86.5	18	1000
Droidbot	45	50.6	82.1	12	1000
Humanoid	45.8	51.2	83.3	12	1000
Sapienz	48.1	53.7	84	16	6000
Stoat	45	50.9	83	14	3000
Monkey	46.8	52.1	80	11	20,000

The results show that adapting Q-Learning with the UCB strategy can significantly improve the effectiveness of the generated test cases. DroidbotX generated a sequence length of 50 events per AUT state with an average of 623 events per run across all apps (which is smaller than the default maximum sequence length of Sapienz). DroidbotX completed exploration before reaching the maximum number of events (set to 1000) within the time limit. Sapienz produced 6000 events and optimized events sequence lengths through the generation of 500 events per AUT state. Nevertheless, it created the largest number of events after Android Monkey. However, the coverage improvement was closer to Humanoid and Droidbot, which generated a smaller number of events. Both Humanoid and Droidbot generated 1000 events per hour. Sapienz uses Android Monkey that requires many events, which may include many redundant events to achieve high coverage. Hence, the coverage gained by Android Monkey only increases slightly as the number of events increases. Thus, a long events sequence length led to a minor positive effect on coverage and crash detection.

Table 5 shows the statistics of models built by Droidbot, Humanoid, and DroidbotX. These tools use the UI transition graph to save the memory of state transitions. The graph model enables DroidbotX to manage the exploration of states systematically to avoid being trapped in a certain state, which also can help to minimize unnecessary transitions. DroidbotX generates an average of 623 events to construct the graph model, while Droidbot and Humanoid generate 969 and 926 average events, respectively. Droidbot cannot exhaustively explore app functions due to its simple exploration strategies. The

depth-first systematic strategy used in Droidbot is surprisingly much less effective than the random strategy since it visits UIs in a fixed order and spends much time on restarting the app when no new UI components are found. Stoat requires more time for test execution due to its model construction in the initial phase which consumes time. Model-free tools such as Android Monkey and Sapienz can easily mislead exploration because of the lack of connectivity information between GUIs [54]. The model constructed by DroidbotX is still not complete since it cannot capture all possible behaviors during exploration, which is still an important research goal on GUI testing [15]. All the events would introduce non-deterministic behavior if they were not properly modeled such as system events and events coming from motion sensors (e.g., accelerometer, gyroscope, and magnetometer). Motion sensors are used for gesture recognition which refers to recognizing meaningful body motions including the movement of the fingers, hands, arms, head, face, or body performed with the intent to convey meaningful information or to interact with the environment [72]. DroidbotX will be extended in the future to include more system events.

Table 5. Statistics of models built by Droidbot, Humanoid, and DroidbotX.

	Droidbot			**Humanoid**			**DroidbotX**		
	Min	**Mean**	**Max**	**Min**	**Mean**	**Max**	**Min**	**Mean**	**Max**
Action	676	969	997	320	926	995	133	623	950
State	9	69	306	8	60	304	10	75	304
Edge	19	171	476	16	127	473	13	177	662

9. Threats to Validity

There are threats and limitations to the validity of our study. Threats to internal validity, the non-deterministic approach of the tools results in obtaining different coverage for each run. Thus, multiple runs were executed to reduce this threat and to remove outliers that could affect the study critically. Each testing tool was allowed to run five times, and the test results were recorded and then computed to yield an average result of final coverage and progressive coverage of the tools. Another threat to the internal validity of our study is Acvtool's instrumentation effect, which affects the integrity of the results obtained. These may be caused by errors triggered by Acvtool's incorrect handling of the binary code or by errors in our experimental scripts. To mitigate this risk, the traces of our experiments for the subject apps were manually inspected.

External validity was threatened by the representativeness of the study to the real world. This means how closely the apps and tools were used in this study to reflect the real world. Moreover, the generalizability of the results was limited as we used a limited number of subject apps. To mitigate these, a standard set of subject apps was used in our experiment from various domains, including fitness, entertainment, and tools applications. The subject apps from F-Droid, which is commonly used in Android GUI testing studies, were carefully selected and the details of the selection process were explained in Section 7.2. Therefore, our test is not prone to selection bias.

10. Conclusions

This research aims to present a Q-Learning-based test coverage approach to generate the GUI test case for Android apps. This approach adopted a UCB exploration strategy to minimize redundant execution of events that improve coverage and crash detection. The proposed approach generated inputs that visit unexplored app states and uses the execution of the app on the generated inputs to construct a state-transition model generated during runtime. This research also provided an empirical evaluation of the effectiveness of the proposed approach and shows a comparison with GUI test-generation tools for Android apps using 30 Android apps. Four criteria (i) instruction coverage, (ii) method coverage, (iii) activity coverage and (iv) number of detected crashes were used to evaluate

and compare GUI test-generation tools. The experimental result revealed that the Q-Learning-based test coverage approach outperforms the state-of-the-art in coverage and in the number of detected crashes within the shortest events sequence length. For future work, DroidbotX will be extended to include input text data, which may integrate text prediction to improve coverage.

Author Contributions: Conceptualization, Data curation, Formal analysis, Methodology, Resources, Software, Visualization, and Writing—Original draft preparation: H.N.Y.; Supervision: S.H.A.H. and R.J.R.Y.; Writing —Review and editing: H.N.Y., S.H.A.H. and R.J.R.Y.; Funding acquisition, R.J.R.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research has received funding from University of Malaya grant numbered RP061E-18SBS, UMRG Program Grant, and European Commission Erasmus plus no: 586297-EPP-1- 2017-1-EL-EPPKA2-CBHE-JP.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available within the article.

Acknowledgments: We gratefully thank University of Malaya for the fundamental research grant provided to us numbered RP061E-18SBS, UMRG Program Grant. Also, this project has been funded by European Commission under the Erasmus plus no: 586297-EPP-1- 2017-1- EL-EPPKA2-CBHE-JP and managed by University of Malaya under grant no. IF024-2018. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Conflicts of Interest: The authors declare that they have no conflict of interest.

References

1. IDC. IDC—Smartphone Market Share—OS. Available online: https://www.idc.com/promo/smartphone-market-share (accessed on 16 December 2019).
2. Chaffey, D. Mobile Marketing Statistics Compilation | Smart Insights. 2018. Available online: https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ (accessed on 16 December 2019).
3. Statista. App Stores. Number of Apps in Leading App Stores 2019 | Statista. Available online: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ (accessed on 14 December 2019).
4. TheAppBrain. Number of Android Applications on the Google Play Store | AppBrain. Available online: https://www.appbrain.com/stats/number-of-android-apps (accessed on 16 December 2019).
5. Packard, H. Failing to Meet Mobile App User Expectations: A Mobile User Survey; Technical Report. 2015. Available online: https://techbeacon.com/sites/default/files/gated_asset/mobile-app-user-survey-failingmeet-user-expectations.pdf (accessed on 16 December 2019).
6. Khalid, H.; Shihab, E.; Nagappan, M.; Hassan, A.E. What do mobile app users complain about? *IEEE Softw.* **2014**, *32*, 70–77. [CrossRef]
7. Martin, W.; Sarro, F.; Harman, M. Causal impact analysis for app releases in google play. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–19 November 2016; pp. 435–446.
8. Ammann, P.; Offutt, J. *Introduction to Software Testing*; Cambridge University Press: Cambridge, UK, 2016.
9. Memon, A. *Comprehensive Framework for Testing Graphical User Interfaces*; University of Pittsburgh: Pittsburgh, PA, USA, 2001.
10. Joorabchi, M.E.; Mesbah, A.; Kruchten, P. Real challenges in mobile app development. In Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 10–11 October 2013; pp. 15–24.
11. Choudhary, S.R.; Gorla, A.; Orso, A. Automated test input generation for android: Are we there yet?(e). In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 429–440.
12. Arnatovich, Y.L.; Wang, L.; Ngo, N.M.; Soh, C. Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customated input generation. *Softw. Pract. Exp.* **2018**, *48*, 1107–1142. [CrossRef]
13. Machiry, A.; Tahiliani, R.; Naik, M. Dynodroid: An input generation system for android apps. In Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 224–234.
14. Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; De Carmine, S.; Memon, A.M. Using GUI ripping for automated testing of Android applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 258–261.

15. Su, T.; Meng, G.; Chen, Y.; Wu, K.; Yang, W.; Yao, Y.; Pu, G.; Liu, Y.; Su, Z. Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 245–256.
16. Mao, K.; Harman, M.; Jia, Y. Sapienz: Multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 94–105.
17. Zhu, H.; Ye, X.; Zhang, X.; Shen, K. A context-aware approach for dynamic gui testing of android applications. In Proceedings of the IEEE 39th Annual Computer Software and Applications Conference, Taichung, Taiwan, 1–5 July 2015; pp. 248–253.
18. Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; Ta, B.D.; Memon, A.M. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Softw.* **2014**, *32*, 53–59. [CrossRef]
19. Wang, W.; Li, D.; Yang, W.; Cao, Y.; Zhang, Z.; Deng, Y.; Xie, T. An empirical study of android test generation tools in industrial cases. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 738–748.
20. Yasin, H.N.; Hamid, S.H.A.; Yusof, R.J.R.; Hamzah, M. An Empirical Analysis of Test Input Generation Tools for Android Apps through a Sequence of Events. *Symmetry* **2020**, *12*, 1894. [CrossRef]
21. Yang, S.; Wu, H.; Zhang, H.; Wang, Y.; Swaminathan, C.; Yan, D.; Rountev, A. Static window transition graphs for Android. *Autom. Softw. Eng.* **2018**, *25*, 833–873. [CrossRef]
22. Memon, A.; Soffa, M.L.; Pollack, M. Coverage criteria for GUI testing. *ACM SIGSOFT Softw. Eng. Notes* **2001**, *26*, 256–267. [CrossRef]
23. Azim, T.; Neamtiu, I. Targeted and depth-first exploration for systematic testing of android apps. In Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, Indianapolis, IN, USA, 26–31 October 2013; pp. 641–660.
24. Koroglu, Y.; Sen, A.; Muslu, O.; Mete, Y.; Ulker, C.; Tanriverdi, T.; Donmez, Y. QBE: QLearning-based exploration of android applications. In Proceedings of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Luxembourg, 18–22 March 2013; pp. 105–115.
25. Yang, S.; Huang, S.; Hui, Z. Theoretical Analysis and Empirical Evaluation of Coverage Indictors for Closed Source APP Testing. *IEEE Access* **2019**, *7*, 162323–162332. [CrossRef]
26. Dashevskyi, S.; Gadyatskaya, O.; Pilgun, A.; Zhauniarovich, Y. The influence of code coverage metrics on automated testing efficiency in android. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2216–2218.
27. Google. UI/Application Exerciser Monkey | Android Developers. Available online: https://developer.android.com/studio/test/monkey (accessed on 10 December 2019).
28. Li, Y.; Yang, Z.; Guo, Y.; Chen, X. DroidBot: A lightweight UI-guided test input generator for Android. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20–28 May 2017; pp. 23–26.
29. Li, Y.; Yang, Z.; Guo, Y.; Chen, X. A Deep Learning based Approach to Automated Android App Testing. *arXiv* **2019**, arXiv:1901.02633.
30. F-Droid. F-Droid—Free and Open Source Android App Repository. Available online: https://f-droid.org/ (accessed on 10 December 2019).
31. Anand, S.; Naik, M.; Harrold, M.J.; Yang, H. Automated concolic testing of smartphone apps. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Cary, NC, USA, 11–16 November 2012; p. 59.
32. Memon, A. GUI testing: Pitfalls and process. *Computer* **2002**, *35*, 87–88. [CrossRef]
33. Yu, S.; Takada, S. Mobile application test case generation focusing on external events. In Proceedings of the 1st International Workshop on Mobile Development, Amsterdam, The Netherlands, 30 October–31 December 2016; pp. 41–42.
34. Rubinov, K.; Baresi, L. What Are We Missing When Testing Our Android Apps? *Computer* **2018**, *51*, 60–68. [CrossRef]
35. Deng, L.; Offutt, J.; Samudio, D. Is mutation analysis effective at testing android apps? In Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25–29 July 2017; pp. 86–93.
36. Kaelbling, L.P.; Littman, M.L.; Moore, A.W. Reinforcement learning: A survey. *J. Artif. Intell. Res.* **1996**, *4*, 237–285. [CrossRef]
37. Watkins, C.J.; Dayan, P. Q-learning. *Mach. Learn.* **1992**, *8*, 279–292. [CrossRef]
38. Google. Understand the Activity Lifecycle | Android Developers. Available online: https://developer.android.com/guide/components/activities/activity-lifecycle.html (accessed on 25 December 2019).
39. Mariani, L.; Pezze, M.; Riganelli, O.; Santoro, M. Autoblacktest: Automatic black-box testing of interactive applications. In Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, 17–21 April 2012; pp. 81–90.
40. Esparcia-Alcázar, A.I.; Almenar, F.; Martínez, M.; Rueda, U.; Vos, T. Q-learning strategies for action selection in the TESTAR automated testing tool. In Proceedings of the 6th International Conferenrence on Metaheuristics and Nature Inspired Computing (META 2016), Marrakech, Morocco, 27–31 October 2016; pp. 130–137.
41. Kim, J.; Kwon, M.; Yoo, S. Generating test input with deep reinforcement learning. In Proceedings of the IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST), Gothenburg, Sweden, 28–29 May 2018; pp. 51–58.

42. Vuong, T.A.T.; Takada, S. A reinforcement learning based approach to automated testing of Android applications. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, Lake Buena Vista, FL, USA, 5 November 2018; pp. 31–37.
43. Adamo, D.; Khan, M.K.; Koppula, S.; Bryce, R. Reinforcement learning for Android GUI testing. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, Lake Buena Vista, FL, USA, 5 November 2018; pp. 2–8.
44. Gu, T.; Cao, C.; Liu, T.; Sun, C.; Deng, J.; Ma, X.; Lü, J. Aimdroid: Activity-insulated multi-level automated testing for android applications. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 103–114.
45. Chen, T.Y.; Kuo, F.-C.; Merkel, R.G.; Tse, T. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.* **2010**, *83*, 60–66. [CrossRef]
46. Mahmood, R.; Mirzaei, N.; Malek, S. Evodroid: Segmented evolutionary testing of android apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 11 November 2014; pp. 599–609.
47. Clapp, L.; Bastani, O.; Anand, S.; Aiken, A. Minimizing GUI event traces. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 422–434.
48. Zheng, H.; Li, D.; Liang, B.; Zeng, X.; Zheng, W.; Deng, Y.; Lam, W.; Yang, W.; Xie, T. Automated test input generation for android: Towards getting there in an industrial case. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Buenos Aires, Argentina, 20–28 May 2017; pp. 253–262.
49. Hu, C.; Neamtiu, I. Automating GUI testing for Android applications. In Proceedings of the 6th International Workshop on Automation of Software Test, Waikiki, Honolulu, HI, USA, 23–24 May 2011; pp. 77–83.
50. Haoyin, L. Automatic android application GUI testing—A random walk approach. In Proceedings of the International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), Chennai, India, 22–24 March 2017; pp. 72–76.
51. Baek, Y.-M.; Bae, D.-H. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 238–249.
52. Yang, W.; Prasad, M.R.; Xie, T. A grey-box approach for automated GUI-model generation of mobile applications. In Proceedings of the International Conference on Fundamental Approaches to Software Engineering, Rome, Italy, 16–24 March; pp. 250–265.
53. Hao, S.; Liu, B.; Nath, S.; Halfond, W.G.; Govindan, R. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, Bretton Woods, NH, USA, 16–19 June 2014; pp. 204–217.
54. Gu, T.; Sun, C.; Ma, X.; Cao, C.; Xu, C.; Yao, Y.; Zhang, Q.; Lu, J.; Su, Z. Practical GUI testing of Android applications via model abstraction and refinement. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 269–280.
55. Adamsen, C.Q.; Mezzetti, G.; Møller, A. Systematic execution of android test suites in adverse conditions. In Proceedings of the International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 13–17 July 2015; pp. 83–93.
56. Moran, K.; Linares-Vásquez, M.; Bernal-Cárdenas, C.; Vendome, C.; Poshyvanyk, D. Automatically discovering, reporting and reproducing android application crashes. In Proceedings of the IEEE International Conference on Software Testing, Verification And Validation (ICST), Chicago, IL, USA, 11–15 April 2016; pp. 33–44.
57. Hu, G.; Yuan, X.; Tang, Y.; Yang, J. Efficiently, effectively detecting mobile app bugs with appdoctor. In Proceedings of the Ninth European Conference on Computer Systems, Amsterdam, The Netherlands, 14–16 April 2014; p. 18.
58. Mirzaei, N.; Bagheri, H.; Mahmood, R.; Malek, S. Sig-droid: Automated system input generation for android applications. In Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, MD, USA, 2–5 November 2015; pp. 461–471.
59. Mariani, L.; Pezzè, M.; Riganelli, O.; Santoro, M. AutoBlackTest: A tool for automatic black-box testing. In Proceedings of the 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; pp. 1013–1015.
60. Lonza, A. *Reinforcement Learning Algorithms With Python: Learn, Understand, and Develop Smart Algorithms for Addressing AI Challenges*; Packt Publishing: Birmingham, UK, 2019.
61. Kamiura, M.; Sano, K. Optimism in the face of uncertainty supported by a statistically-designed multi-armed bandit algorithm. *Biosystems* **2017**, *160*, 25–32. [CrossRef] [PubMed]
62. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2011.
63. Bauersfeld, S.; Vos, T.E. User interface level testing with TESTAR; what about more sophisticated action specification and selection? In Proceedings of the 7th Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE 2014), L'Aquila, Italy, 9–11 July 2014; pp. 60–78.
64. Choi, W.; Necula, G.; Sen, K. Guided gui testing of android apps with minimal restart and approximate learning. In Proceedings of the ACM Sigplan International Conference on Object Oriented Programming Systems Languages & Applications, Indianapolis, IN, USA, 29 October 2013; pp. 623–640.
65. Kitchenham, B.A.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; El Emam, K.; Rosenberg, J. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.* **2002**, *28*, 721–734. [CrossRef]

66. Perry, D.E.; Sim, S.E.; Easterbrook, S.M. Case studies for software engineers. In Proceedings of the 26th International Conference on Software Engineering, Edinburgh, UK, 23–28 May 2004; pp. 736–738.
67. Freke, J. Smali, an Assembler/Disassembler for Android's Dex Format. Available online: https://github.com/JesusFreke/smali (accessed on 26 December 2020).
68. Huang, C.-Y.; Chiu, C.-H.; Lin, C.-H.; Tzeng, H.-W. Code coverage measurement for Android dynamic analysis tools. In Proceedings of the 4th IEEE International Conference on Mobile Services (MS 2015), New York, NY, USA, 27 June–2 July 2015; pp. 209–216.
69. Google. Crashes | Android Developers. Available online: https://developer.android.com/topic/performance/vitals/crash (accessed on 25 December 2019).
70. Pilgun, A.; Gadyatskaya, O.; Zhauniarovich, Y.; Dashevskyi, S.; Kushniarou, A.; Mauw, S. Fine-grained code coverage measurement in automated black-box Android testing. *ACM Trans. Softw. Eng. Methodol.* **2020**, *29*, 1–35. [CrossRef]
71. Google. Command Line Tools | Android Developers. Available online: https://developer.android.com/studio/command-line (accessed on 16 December 2019).
72. Azmi, S.S.; Yusof, R.J.R.; Chiew, T.K.; Geok, J.C.P.; Sim, G. Gesture Interfacing for People with Disability of the Arm, Shoulder and Hand (Dash) for Smart Door Control: Goms Analysis. *Malays. J. Comput. Sci.* **2019**, 98–117. [CrossRef]