Reinforcement Learning for Android GUI Testing

David Adamo
Ultimate Software Group, Inc.
Weston, Florida 33326, USA
david_adamo@ultimatesoftware.com

Md Khorrom Khan
University of North Texas
Denton, Texas 76207, USA
mdkhorromkhan@my.unt.edu

Sreedevi Koppula
University of North Texas
Denton, Texas 76207, USA
sreedevikoppula@my.unt.edu

Renée Bryce
University of North Texas
Denton, Texas 76207, USA
renee.bryce@unt.edu

ABSTRACT

This paper presents a reinforcement learning approach to automated GUI testing of Android apps. We use a test generation algorithm based on Q-learning to systematically select events and explore the GUI of an application under test without requiring a preexisting abstract model. We empirically evaluate the algorithm on eight Android applications and find that the proposed approach generates test suites that achieve between 3.31% to 18.83% better block-level code coverage than random test generation.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; **Software verification and validation**; **Software defect analysis**; **Software testing and debugging**;

KEYWORDS

GUI Testing, Mobile application testing, Android, Q-learning

ACM Reference Format:
David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. Reinforcement Learning for Android GUI Testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '18), November 5, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3278186.3278187

1 INTRODUCTION

The Google Android platform dominates the mobile OS market with 85% of the Smartphone OS Market Share [16]. End users often interact with Android applications through a Graphical User Interface (GUI). An application's failure to respond appropriately to GUI actions often results in its inability to fulfill the intent of a user. GUI testing can be done either by manual interaction or automation with test scripts. Owing to the potentially large behavior space as well as the high level of human involvement, these approaches become difficult to scale. A study shows that of over 600 open-source Android application projects, only 14% contain test cases and approximately 9% have executable test cases with above 40% code coverage [17]. Even though there is existing research [3, 9, 10, 15, 19] toward developing automated GUI testing techniques for Android applications, there is still a need for continued research due to the low code coverage of existing techniques.

Android applications are built around unique components of the Android development framework. Users interact with Android applications through various input methods including hardware keyboards, software keyboards and touchscreens. Android supports a wide array of gestures such as tap, drag, slide, pinch and rotate. We need to consider these interaction mechanisms and the unique architecture of the Android framework to develop automated testing techniques for Android applications. Random test generation is one popular technique for testing Android applications [13, 19, 21, 27]. A simple approach is to feed random GUI events to the Application Under Test (AUT) during its execution. This approach has some limitations. Parts of the application that require more thorough testing may not receive proportionate attention in relation to parts that have already been explored. This problem is even more pronounced when the AUT requires repeated execution of specific event sequences in order to access unexplored parts of the application.

This paper uses reinforcement learning and event selection heuristics to systematically generate test suites with improved code coverage relative to random test generation. The proposed Q-learning-based technique uses trial-and-error interactions to identify events that are likely to discover unexplored states and revisit partially explored states. In a particular state, the Q-learning agent selects an event from the available set of events based on associated rewards derived from past interactions. Previously unexplored events have higher rewards than already explored events and have higher priority to be selected next. This paper makes the following main contributions:

- An adaptation of reinforcement learning to automated GUI testing of Android applications with a Q-learning-based test generation algorithm.
- An empirical evaluation that compares the Q-learning-based algorithm to random test generation in terms of block coverage across eight Android applications.

We use reinforcement learning techniques to address the aforementioned limitations of random test generation within the context of GUI-based Android applications. Our test generation algorithm intelligently selects events from the GUI during application execution and uses dynamic event extraction [5] without need for a preexisting abstract model of AUT. We hypothesize that parts of an AUT

A-TEST '18, November 5, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6053-1/18/11…$15.00
https://doi.org/10.1145/3278186.3278187

that require extra attention are likely to be tested more thoroughly with Q-learning [25]. We discuss the proposed Q-learning-based technique in detail in section 3.

Structure of the Paper. Section 2 provides background on Q-learning and Android automated testing. Section 3 presents our proposed approach to test generation. Section 4 presents an empirical evaluation of our Q-learning-based technique on eight Android applications. Section 5 concludes with a discussion of current and future work.

2 BACKGROUND AND RELATED WORK

Developers often perform limited testing due to the complexity of modern mobile applications and time-to-market pressure. Researchers have developed techniques to automate Android GUI testing. This section highlights the shortcomings of existing techniques and provides background information on reinforcement learning and its application to automated GUI testing.

2.1 Automated GUI Testing

Model-based techniques [6, 14, 24] rely on abstract models of an AUT as the basis for test generation. Amalfitano et al. [3] created *AndroidRipper* to dynamically construct a model of the AUT as a GUI tree and then select paths through the tree to generate test cases. Espada et al. [10] use a model checking approach which composes state machines to model the behaviors of the AUTs. A model checker generates execution traces from the model which correspond to test cases. One major drawback of model-based techniques is that they have a tendency to generate infeasible test cases that cannot be executed on the AUT. Choi et al. [9] use active learning to construct a model of an AUT. Their technique improves code coverage by avoiding application restarts during the model learning process. This work focuses on optimizing event selection to reach unexplored areas of the AUT.

Monkey [21] is a popular random GUI testing tool that is part of the Android Software Development Framework (SDK). Monkey generates GUI tests by interacting with random screen coordinates on Android devices. Cuixiong Hu et al. [15] combine automatic test generation using Monkey with dynamic analysis of log files. Monkey-based techniques do not limit test generation to relevant events in the AUT and often generate a large number of events that do not expose any behavior.

Dynamic event extraction techniques do not use a preexisting abstract model to generate test cases. These techniques identify, select and execute relevant events from the AUT's GUI at runtime to generate event sequences. Dynamic event extraction techniques do not generate infeasible test cases since the test cases are based on runtime interaction with the AUT [5]. Machiry et al. [19] present Dynodroid, an input generation system for Android applications. Dynodroid uses random and frequency-based algorithms to dynamically generate GUI tests for Android applications.

This work adopts a dynamic event extraction approach similar to that of Dynodroid. We use reinforcement learning techniques to optimize event selection in an attempt to improve code coverage relative to random test generation.

2.2 Reinforcement Learning

Reinforcement learning [25, 26] is a computational approach to machine learning, inspired by behavioral psychology and focused on goal-directed learning from interactions. It directly connects an action with an outcome to learn about the best actions based on reward or punishment. The two main components of a reinforcement learning set up are the agent and the environment. The agent is an autonomous entity that can perform independent actions in an environment to achieve a goal. The object the agent acts on is considered to be the environment. The agent uses trial-and-error interactions to gain information about the environment. Some other basic concepts in reinforcement learning include *state*, *action*, *reward*, and *policy*. *State* describes the current situation of the environment. An *action* is a possible move in a particular state. Reward is an abstract concept to evaluate actions, i.e., the immediate feedback after performing an action. *Policy* defines the strategy that an agent uses to select an action from a given state.

This work uses Q-learning, a reinforcement learning technique that enables an agent to learn an action selection policy in an environment through trial-and-error interactions [25]. The agent chooses an action based on the current state of the environment and evaluates the resulting reward. The goal of the agent is to learn a sequence of actions that maximizes cumulative reward. In the context of automated GUI testing, the testing tool is the agent, the AUT is the environment and GUI events represent actions in each state. The testing tool applies trial and error interactions to the AUT to find a policy that facilitates systematic exploration of the GUI. This exploration process generates event sequences that can be used as test cases.

2.3 Automated GUI Testing with Q-learning

Mariani et al. [20] describes AutoBlackTest, an automated GUI testing tool for desktop applications. The tool implements an algorithm that uses Q-learning to interact with a Java/Swing desktop application and explore its GUI. The implementation of AutoBlackTest in [20] supports only Java/Swing applications and does not provide an implementation or evaluation of Q-learning that considers the unique characteristics of mobile applications. Bauersfeld and Vos [7] propose an action specification and selection mechanism based on Prolog specifications and Q-learning for user interface-level testing. The idea is to change the probability distribution over the event sequence space at runtime in order to favor exploration of the GUI. The approach was implemented for and evaluated on Mac OS X desktop applications. The empirical evaluations by Mariani et al. [20] and Bauersfeld and Vos [7] do not include a comparison with random test generation as suggested by Arcuri et al. [4] for such empirical studies. Carino [8] evaluated multiple algorithms to dynamically test Java desktop applications including a Q-learning based algorithm. The techniques in [8] are not directly applicable to Android applications due to the significant difference between Java desktop applications and the Android framework. Tianxiao et al. [12] describe AimDroid, an Android GUI testing tool that uses a reinforcement learning technique called SARSA [25] to guide exploration of Android applications.

We extend and adapt the work in Mariani et al. [20] to automated GUI testing of Android applications. We describe a Q-learning-based technique to automatically explore the GUI of Android applications and generate test suites with improved code coverage relative to random test generation.

3 PROPOSED APPROACH

3.1 Representation of States and Events

We use Q-learning for test generation where the AUT represents the agent's environment. In each state, the agent chooses an event, executes the chosen event and evaluates the resulting reward. Each state contains information that helps the agent to select the next event. The agent's goal is to generate event sequences that maximize cumulative reward. The definitions of what comprises a state and an event are crucial to the reinforcement learning approach described in this paper. The ability to deterministically identify each state and event enables the agent to make event selection decisions across different runs. During test generation, we use Appium [11] and UIAutomator [22] to retrieve XML representations of the AUT's GUI. The XML representation enables the agent to discover the types of widgets available in each GUI state and the types of actions (e.g. click, long press, etc.) that are enabled on the widgets. Widgets are uniquely identified by ID or XPath depending on what is available. This information provides the basis for our definition of states, actions and events.

Definition 3.1. An action a is denoted by a 3-tuple: $a = (w, t, v)$, where w is a widget on a particular screen, t is a type of action that can be performed on the widget (e.g. click) and v holds arbitrary text if the widget w is a text field. For all non-text field widgets, the value of v is empty.

Definition 3.2. A GUI state s is denoted by an n-tuple:
$s = (a_1, a_2, a_3, ..., a_n)$ where a_i is an action and n is the total number of unique actions available on the screen.

Definition 3.3. An event is a set of one or more related actions that may occur in a particular GUI state. It is denoted by a 2-tuple $e = (s, A_e)$ where s is a GUI state and A_e is an ordered set of actions associated with the event.

3.2 Reward Function

We model an AUT as a stochastic process with a finite set of GUI states S and a finite set of GUI events E (e.g., tapping a widget, text input, etc.). For a given GUI state $s \in S$, the test generation agent selects and executes an event $e \in E$ from the set of available events in s. Event selection is based on a notion of reward and event execution may cause a transition to a new GUI state s'. A reward function calculates the reward associated with a given GUI event. In this work, we define the reward R for taking event e in GUI state s which leads to state s', as follows:

$$R(e, s, s') = \frac{1}{x_e} \tag{1}$$

where x_e is the number of times event e has been executed in GUI state s. The reward for executing an event is inversely proportional to the number of times it has been executed in the past. The reward function assigns its highest reward when an event is executed for the first time.

3.3 Q-value Function

The Q-value function is based on the immediate reward for executing an event and the expected future reward associated with subsequent states. The choice of which event to select in a particular state is influenced not only by the immediate reward for executing the event but also by potential rewards from events in future states. The Q-value function enables the test generation agent to "look ahead" when making the choice of what event to select in a particular state. Sometimes immediate sacrifice may result in higher future rewards in the long run. The Q-value function is recursively defined as follows:

$$Q(s, e^*) = R(e^*, s, s') + \gamma \cdot max_{e \in E_{s'}} Q(s', e) \tag{2}$$

where $Q(s, e^*)$ is the Q-value of event e^* in state s, $R(e^*, s, s')$ is the immediate reward for executing event e^* in state s, $max_{e \in E_{s'}} Q(s', e)$ is the maximum Q-value in state s' that results from taking event e^* in state s and γ is a parameter called the *discount factor*.

The test generation agent iteratively approximates the Q-value of GUI events based on its experience interacting with the AUT. The Q-value function enables the test generation agent to favor execution of GUI events that lead to unexplored and/or partially explored states, irrespective of immediate reward. In our implementation, the Q-value of each event e^* is initialized to a user-defined default value. Each time an event e^* in a particular GUI state is selected, the Q-value is updated using equation 2.

3.4 Discount Factor

The value of the discount factor γ for a state s' with available events E is given by equation 3.

$$\gamma(s', E) = 0.9 \times e^{-0.1 \times (|E|-1)} \tag{3}$$

where $|E|$ is the number of events in state s'. The discount factor controls the extent to which expected future rewards affect the choice of an event in the current GUI state. A high discount factor encourages selection of events that potentially lead to high rewards in future states. A low discount factor prioritizes immediate rewards over long-term rewards. A discount factor of 0 causes the test generation agent to consider only immediate rewards. A discount factor approaching 1 enables the test generation agent to place high priority on cumulative future rewards. We use a dynamic discount factor calculated by the exponential decay function in equation 3 instead of a fixed discount factor across all states. The intuition is that the agent should look further ahead (i.e. use a high discount value) and prioritize potential future rewards over immediate rewards when it encounters states with a small number of events. The dynamic discount factor as defined in equation 3 also reduces the tendency of the agent to ignore states that have a small number of available events.

3.5 Test Generation

The test generation agent selects events and generates event sequences in a manner that maximizes cumulative reward. In each state, the agent chooses an event that has the highest Q-value from the set of available events. Algorithm 1 shows pseudocode for our Q-learning-based test generation algorithm. It takes four input parameters: 1) application under test, 2) test suite completion criterion, 3) probability of HOME button and 4) initial Q-value for new events. The algorithm is part of an Android test generation tool called Autodroid [1, 2]. It uses the input parameters to explore the GUI and produces a set of event sequences as a test suite for the AUT.

Algorithm 1: Test Suite Generation

```
    input  : application under test, $AUT$
    input  : test suite completion criterion, $c$
    input  : home button probability, $home\_btn\_prob$
    input  : initial Q-value, $V_{init}$
    output : test suite, $T$
1   begin
2       while not c do
3           start AUT;
4           $testCase \leftarrow \phi$;
5           while true do
6               if random(0, 1) <= $home\_btn\_prob$ then
7                   $selectedEvent \leftarrow HOME$
8               else
9                   $currEvents \leftarrow$ getAvailableEvents();
10                  foreach $event$ in $currEvents$ do
11                      if timesExecuted($event$) = 0 then
12                          setQValue($event, V_{init}$);
13                      end
14                  end
15                  $selectedEvent \leftarrow$ getMaxValueEvent()
16              end
17              execute $selectedEvent$;
18              $testCase \leftarrow testCase \cup selectedEvent$;
19              if $selectedEvent$ exits AUT then
20                  updateReward($selectedEvent, 0$);
21                  setQValue($selectedEvent, 0$);
22                  break;
23              end
24              $newEvents \leftarrow$ getAvailableEvents();
25              $\gamma \leftarrow$ calDiscountFactor(newEvents);
26              $reward \leftarrow$ getReward($selectedEvent$);
27              $maxValue \leftarrow$ getMaxValue($newEvents$);
28              $qValue \leftarrow reward + \gamma \times maxValue$;
29              setQValue($selectedEvent, qValue$);
30          end
31          $T \leftarrow T \cup testCase$;
32      end
33  end
```

The criterion for test suite completion is a fixed time budget. On each iteration, the algorithm creates an empty test case and starts the AUT. The *getAvailableEvents* procedure on line 9 gets all the events available in the current GUI state at the time it is called. Lines 10-14 set the initial Q-value to the user-specified value V_{init} for events that have never been executed. The *getMaxValueEvent* procedure on line 15 selects the event that has the maximum Q-value from the events in the current GUI state and line 17 executes the selected event. The call to *getAvailableEvents* on line 24 gets the available events in the GUI state resulting from executing a selected event in the previous state. Lines 25-26 calculate the reward and discount factor for the executed event as defined in equations 1 and 3 respectively. The *getMaxValue* procedure on line 27 returns the maximum Q-value in the resulting state. This value represents an estimate of future rewards that may be accrued by executing the selected event. Each time an event is executed, it is added to the event sequence for the current test case and the Q-value estimate is updated on line 29 using equation 2.

The event-selection and Q-value update process repeats until the agent executes a termination event that causes the AUT to close. The events executed until the termination point represent a single test case. Examples of termination events include: (i) events whose sole purpose is to exit the application (e.g. clicking an exit button in a menu), (ii) pressing the HOME button, (iii) pressing the BACK button in certain GUI states, (iv) GUI events that cause a switch to some other application e.g. the Android contacts application and (v) events that cause the AUT to crash.

Lines 19-23 assign a Q-value of zero to termination events. This enables the test generation agent to avoid previously encountered termination events that may prevent deeper exploration of the GUI in subsequent test cases. This blacklisting scheme also helps to prevent generation of a high number of short test cases. It does not apply to the Android HOME button since the HOME button is used solely to probabilistically terminate each test case as shown on lines 6-7. Probabilistic termination of event sequences enables the agent to produce test cases of varying length within a test suite. We use an event sequence hash function to prevent generation of duplicate test cases so that every test case in a test suite is unique.

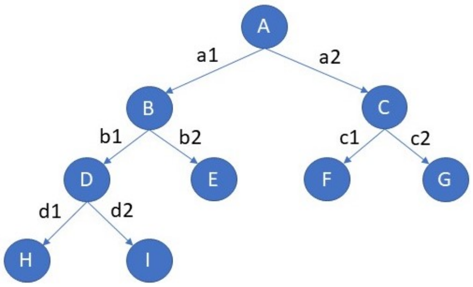

Figure 1: Example application in terms of states and events

3.6 Example

Consider generation of a test suite for an Android application that has states A, B, C, D, E, F, G, H and I as shown in Figure 1. State A has two available events ($a1$ and $a2$), state B has two available events ($b1$ and $b2$) and state C has two available events ($c1$ and $c2$). States E, F, G, H and I are leaf nodes that represent the result of executing a termination event.

The initial Q-value for each event is set to 500. The directed arrow indicates a transition from one state to another upon execution of the event indicated by the arrow's label. For instance, executing event $a1$ causes a state transition from A to B. As shown in Algorithm 1, app execution starts at line 3 and the number of events in the *testCase* is initially 0. We consider this as the start of an episode.

At the start of episode 0, the agent is in state A and line 9 of the algorithm sets the *currEvents* value to {$a1$, $a2$}. Lines 10-14 set the Q-value for $a1$ and $a2$ to the user-specified initial Q-value of 500. Line 15 selects an event that has the highest Q-value in the current GUI state. Since both $a1$ and $a2$ have the same Q-value, one of them is selected randomly. If we assume that event $a2$ is selected, then line 17 executes the *selectedEvent* $a2$. This causes a transition from state A to state C and the execution count of $a2$ is updated to 1 (initial execution count was 0). Line 18 adds the executed event $a2$ to the *testCase*. If the executed event closes the application, lines 19-23 set the reward and Q-value for the event to 0. Event $a2$ does not close the application, so line 24 sets the value of *newEvents* to {c1, c2}. Line 25 calculates the discount factor using equation 3. Line 26 calculates the reward for executing event $a2$ as defined in equation 1 and line 27 gets the maximum Q-value of the events $c1$ and $c2$ which is 500 (both events still have the same initial Q-value since they have never been executed). Lines 28-29 calculates and sets the new Q-value for the executed event $a2$. In

Table 1: Example rewards and Q-values for three episodes

state	event	Episode 0			Episode 1			Episode 2		
		Count	**Reward**	**Q-value**	**Count**	**Reward**	**Q-value**	**Count**	**Reward**	**Q-value**
A	a1	0	-	500	1	1	408	1	1	408
	a2	1	1	408	1	1	408	2	0.5	407.5
B	b1	0	-	500	0	-	500	0	-	500
	b2	0	-	500	1	0	0	1	0	0
C	c1	1	0	0	1	0	0	1	0	0
	c2	0	-	500	0	-	500	1	0	0
D	d1	0	-	500	0	-	500	0	-	500
	d2	0	-	500	0	-	500	0	-	500

state C, both events have the same Q-value. Suppose $c1$ is selected randomly and executed, then a transition is made from state C to F. The reward and Q-value for $c1$ is set to 0 since state F is a terminal state. The agent repeats this process for each episode until it executes a termination event that closes the AUT. Application exit indicates the end of an episode/test case and the resulting test case is added to the test suite. Table 1 shows the reward and Q-value for each event after each episode.

In episode 1, the agent uses information derived from episode 0. In state A, it selects and executes event $a1$ since it has the largest Q-value. This causes a transition from state A to state B. In state B, both $b1$ and $b2$ have the same Q-value. If we assume $b2$ is selected, then a transition from state B to E occurs. Since E is a terminal state, the reward and Q-value for $b2$ is set to 0 and the application closes. The updated reward and Q-value for each event in episode 1 is shown in Table 1 under the column "Episode 1". The algorithm continues to generate test cases until it meets the specified test suite completion criteria.

4 EMPIRICAL EVALUATION

We evaluate our Q-learning based technique by comparing its performance to random test generation in terms of code coverage across eight subject applications. The goal is to answer the following research question:

Research Question. *Does the Q-learning-based algorithm generate test cases with higher code coverage than random test generation?*

Table 2: Characteristics of subject applications

App Name	# Lines	# Methods	# Classes	# blocks
Tomdroid v0.7.2	5736	496	131	22169
Loaned v1.0.2	2837	258	70	9781
Budget v4.0	3159	367	67	9129
ATimeTracker v0.23	1980	130	22	8351
Repay v1.6	2059	204	48	7124
SimpleDo v1.2.0	1259	88	31	5355
Moneybalance v1.0	1460	163	37	4959
WhoHasMyStuff v1.0.25	1026	90	24	3597

4.1 Applications Under Test

We evaluate our Q-learning-based technique on eight Android applications. These applications are from multiple categories in the F-droid open source repository [18] and they serve different purposes. *Tomdroid* is a note-taking application. *Budget* is an application to manage income and expenses. *ATimeTracker* helps users to start/stop time tracking for any task. *Moneybalance* tracks expenses shared by groups of people. *WhoHasMyStuff* and *Loaned* are inventory apps to keep track of personal items. *Repay* is an app to keep track of debts. *SimpleDo* is a todo list application.

Table 2 shows characteristics of the subject applications including number of lines, methods, classes and bytecode blocks in each application. The applications range from 1026 to 5736 lines of code, 88 to 496 methods, 22 to 131 classes and 3597 to 22169 bytecode blocks. *Tomdroid* has the largest number of code lines and bytecode blocks with 5736 and 22169 respectively. *WhoHasMyStuff* has the smallest number of code lines and bytecode blocks with 1096 and 3597 respectively. The applications contain a variety of input controls such as buttons, checkboxes, radio buttons, spinners, pickers, options menu, floating contextual menus, pop-up menus, and dialog boxes. We instrumented the bytecode of each Android application using the techniques described in Zhauniarovich et al. [28].

4.2 Experimental Setup

We use random test generation as a baseline to evaluate the performance of our Q-learning test generation technique. Random test generation selects and executes events uniformly at random from the available events in each GUI state. We implement the random and Q-learning-based test generation algorithms in the same tool, Autodroid, to minimize the influence of different tool implementations on the results of the experiment. Autodroid takes instrumented APK files as input to generate test suites and code coverage reports. Code coverage reports are generated using Emma [23]. We generated the test cases on Android 4.4 emulators with screen resolution 768x1280. We ran the emulators on an Ubuntu 14.04 host machine with 16GB RAM. Table 3 shows the configuration parameters used to instantiate the random test generation and Q-learning-based algorithms.

We run each test generation algorithm for 2 hours on each application to generate a test suite. Based on the size of the applications, we assume that 2 hours of testing time is reasonable for covering most of the application features. We run both test generation algorithms 10 times on each subject application to minimize the impact of randomness in the algorithms. We use a time delay of 4 seconds between events so that the AUT has sufficient time to respond to one event before performing the next one. The probability of pressing the HOME button in a GUI state influences the average length of test cases in a test suite since the HOME button causes the AUT to exit. We set the HOME button probability to 5% since prior experiments suggest that the 5% probability value provides a reasonable balance between short and long test cases. For the Q-learning algorithm, we use a high initial Q-value of 500 for events that have never been executed. This value is larger than any Q-value the test generation agent will derive from actual interaction with the AUT. Such a high initial Q-value encourages the

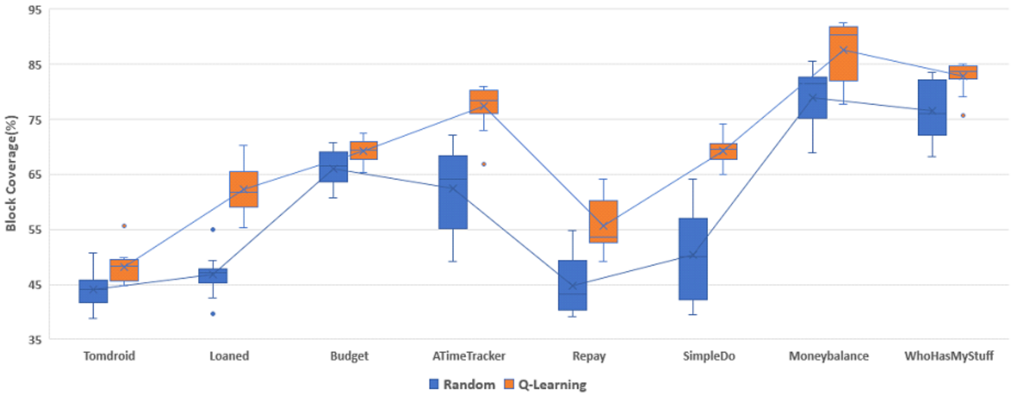

Figure 2: Block coverage across all applications and all runs

Table 3: Test generation parameters

Parameters	Random	Q-learning
Generation time for each test suite (in hours)	2	2
Number of test suites (trials) for each app	10	10
Time delay between actions (in seconds)	4	4
Home button probability	0.05	0.05
Initial Q-value	-	500

test generation agent to execute each event in a GUI state at least once before making decisions based on learned Q-values. The high initial Q-value of 500 also encourages repeated execution of event sequences that revisit partially explored states (i.e states with at least one event that has never been executed).

4.3 Results and Discussion

We use code coverage as a measure of the extent to which each test generation technique explores the functionality of the AUT. Instead of comparing results separately for each application, we normalize and compare the combined data across all the applications for each technique. In line with recommendations in Arcuri et al. [4], we use Mann-Whitney U-test for pairwise statistical tests.

Table 4 shows the average block coverage achieved by random test generation and our Q-learning-based technique. The Q-learning technique achieves higher average block coverage for all applications in our experiments and shows an average improvement of 10.30% compared to random test generation. The difference between average block coverage achieved by the random and Q-learning-based techniques ranges from 3.31% to 18.83%.

Table 4: Average block coverage

App	Random	Q-learning	Improvement by Q-learning
Tomdroid	44.06%	**48.20%**	**4.14%**
Loaned	46.78%	**62.28%**	**15.50%**
Budget	65.95%	**69.26%**	**3.31%**
ATimeTracker	62.47%	**77.31%**	**14.84%**
Repay	44.79%	**55.69%**	**10.90%**
SimpleDo	50.41%	**69.24%**	**18.83%**
Moneybalance	78.90%	**87.50%**	**8.60%**
WhoHasMyStuff	76.50%	**82.75%**	**6.25%**

Figure 2 shows a box plot of block coverage achieved by each algorithm for all the subject applications. The Q-learning-based technique has a higher median block coverage compared to random test generation for each of the applications. Q-learning consistently achieves higher maximum coverage than random test generation whereas random test generation always has the lowest coverage for each application. We performed a Mann-Whitney U-test to determine if there is a significant difference between the block coverage achieved by the Q-learning-based and random test generation algorithms. The Mann-Whitney U-test shows that there is a significant difference ($U = 1911.5, p = 0.00001105374$) between the block coverage of both techniques at the p < 0.05 significance level.

The difference in block coverage between the random and Q-learning-based techniques is most notable in *SimpleDo*, *Loaned*, *ATimeTracker* and *Repay*. These apps have GUIs that mostly require simple actions such as clicks and long presses rather than complex interactions with validated text input fields. Furthermore, the majority of their functionality is accessible through a small subset of GUI states. The Q-learning-based algorithm assigns Q-values to encourage execution of events that lead to new or partially explored states. This enables the algorithm to repeatedly execute sequences of high-value events and revisit the subset of GUI states that provide access to most of an AUT's functionality. The block coverage improvement in *Budget*, *Moneybalance*, *Tomdroid*, and *WhoHasMyStuff* is smaller than the other subject applications for a number of possible reasons. *Moneybalance* and *WhoHasMyStuff* have a small number of features, most of which are easily accessible by the random and Q-learning-based algorithms. *Budget* has several GUI states that require complex interactions with validated text input fields. *Tomdroid* has a significant amount of OS-specific and configuration-dependent code that is unreachable regardless of which test generation algorithm is used. The block coverage improvements in these apps may be due, in large part, to the Q-learning algorithm's ability to identify and avoid termination events that prevent deep exploration of the GUI within a single episode.

4.4 Threats to Validity

We compared our Q-learning technique to random test generation across eight open source Android applications. The limited number of subject applications and the comparison with only one alternative technique is a threat to external validity. Further studies that compare the Q-learning-based technique to greedy frequency-based [1] and combinatorial-based [2] alternatives, with a higher number of subject applications, may increase confidence in the results. Nevertheless, the consistency of the results in this initial study suggests that our Q-learning-based technique has some potential value. We evaluated the performance of our technique only in terms of block coverage. Further studies may examine fault detection ability and the impact of different time budgets for applications of different complexity. Selection of input parameter values is an internal threat to validity since these affect the behavior of the Q-learning algorithm. It is difficult to choose parameters that work well in all cases, especially for machine learning algorithms. The choice of discount factor, event selection heuristics, and initial Q-value may have a significant effect on the test generation process and show different results. We used a variable discount factor calculated by an exponential decay function on the basis of our intuition that the agent should look further ahead (i.e. use a high discount value) when it encounters states with a small number of events. Further experimentation is needed in this regard with varying values of configuration parameters.

5 CONCLUSIONS AND FUTURE WORK

Automated test generation for Android applications involves simulation of user interactions. The unique characteristics of the Android framework and the large number of possible events make test generation challenging. This work described and evaluated a Q-learning-based technique for automated GUI testing of Android apps. Our test generation algorithm uses trial-and-error interactions to optimize event selection and systematically explore an AUT's GUI to generate test cases. Empirical evaluation shows that for the same set of test generation parameters (i.e. generation time, delay between events, home button probability, etc.), the Q-learning-based technique achieves 10.30% higher block coverage on average than random test generation, with a range of 3.31% to 18.83% improvement across eight subject applications. Future work will compare the code coverage rates of the random test generation and Q-learning-based algorithms. We will also investigate different event selection heuristics, reward functions, discount factors, initial values, and other reinforcement learning techniques (e.g. Monte Carlo, SARSA, experience replay, etc.) across a larger set of applications.

ACKNOWLEDGMENTS

This work is supported in part by Ultimate Software Group, Inc. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of Ultimate Software or University of North Texas.

REFERENCES

[1] David Adamo, Renée Bryce, and Tariq M. King. 2018. Randomized Event Sequence Generation Strategies for Automated Testing of Android Apps. In *Information Technology - New Generations*, Shahram Latifi (Ed.). Springer International Publishing, Cham, 571–578.

[2] David Adamo, Dmitry Nurmuradov, Shraddha Piparia, and Renée Bryce. 2018. Combinatorial-based event sequence testing of Android applications. *Information and Software Technology* 99 (2018), 98 – 117.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 258–261.

[4] A. Arcuri and L. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*. 1–10.

[5] Gigon Bae, Gregg Rothermel, and Doo-Hwan Bae. 2014. Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study. *Journal of Systems and Software* 97, 15–46.

[6] Young-Min Baek and Doo-Hwan Bae. 2016. 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 (ASE 2016)*. ACM, New York, NY, USA, 238–249.

[7] Sebastian Bauersfeld and Tanja EJ Vos. 2014. User interface level testing with TESTAR; what about more sophisticated action specification and selection?. In *CEUR Workshop Proceedings*, Vol. 1354. 60–78.

[8] Santo Carino. 2016. *Dynamically Testing Graphical User Interfaces.* Ph.D. Dissertation. The University of Western Ontario, London, ON, CA.

[9] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. ACM, New York, NY, USA, 623–640.

[10] Ana Rosario Espada, María del Mar Gallardo, Alberto Salmerón, and Pedro Merino. 2015. Using Model Checking to Generate Test Cases for Android Applications. In *Tenth Workshop on Model-Based Testing (MBT 2015)*. 7–21.

[11] JS Foundation. 2018. Appium: Mobile App Automation Made Awesome. http://appium.io/. Accessed: 2018-07-02.

[12] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lu. 2017. AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 103–114.

[13] L. V. Haoyin. 2017. Automatic android application GUI testing – A random walk approach. In *International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. 72–76.

[14] J. Harty, M. Katara, and T. Takala. 2011. Experiences of System-Level Model-Based GUI Testing of an Android Application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation(ICST)*. 377–386.

[15] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST 2011)*. ACM, New York, NY, USA, 77–83.

[16] IDC Research. 2017. Smartphone OS Market Share, 2017 Q1. https://www.idc.com/promo/smartphone-market-share/os. Accessed: 2018-02-21.

[17] Pavneet Singh Kochhar, Thung Ferdian, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the Test Automation Culture of App Developers. In *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[18] F-Droid Limited and Contributors. 2018. F-Droid - Free and Open Source Android App Repository. https://f-droid.org/. Accessed: 2018-02-10.

[19] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234.

[20] Leonardo Mariani, Mauro Pezzé, Oliviero Riganelli, and Mauro Santoro. 2014. Automatic Testing of GUI-based Applications. *Software Testing, Verification and Reliability* 24, 5 (2014), 341–366.

[21] Android Open Source Project. 2018. Android UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html. Accessed: 2018-07-02.

[22] Android Open Source Project. 2018. UIAutomator. https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html. Accessed: 2018-07-02.

[23] Vlad Roubtsov. 2005. Emma. http://emma.sourceforge.net/. Accessed: 2018-03-12.

[24] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.

[25] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction.* MIT press.

[26] Martijn van Otterlo and Marco Wiering. 2012. Reinforcement Learning and Markov Decision Processes. In *Reinforcement Learning: State-of-the-Art*, Marco Wiering and Martijn van Otterlo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–42.

[27] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing and Multimedia (MoMM '13)*. ACM, New York, NY, USA, 68–74.

[28] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci. 2015. Towards Black Box Testing of Android Apps. In *2015 10th International Conference on Availability, Reliability and Security*. IEEE, 501–510.