A Reinforcement Learning Based Approach to Automated Testing of Android Applications

Thi Anh Tuyet Vuong
Grad. School of Science and Technology, Keio University
Yokohama, Japan
tuyet@doi.ics.keio.ac.jp

Shingo Takada
Grad. School of Science and Technology, Keio University
Yokohama, Japan
michigan@ics.keio.ac.jp

ABSTRACT

In recent years, researchers have actively proposed tools to automate testing for Android applications. Their techniques, however, still encounter major difficulties. First is the difficulty of achieving high code coverage because applications usually have a large number of possible combinations of operations and transitions, which makes testing all possible scenarios time-consuming and ineffective for large systems. Second is the difficulty of achieving a wide range of application functionalities, because some functionalities can only be reached through a specific sequence of events. Therefore they are tested less often in random testing. Facing these problems, we apply a reinforcement learning algorithm called Q-learning to take advantage of both random and model-based testing. A Q-learning agent interacts with the Android application, builds a behavioral model gradually and generates test cases based on the model. The agent explores the application in an optimal way that reveals as much functionalities of the application as possible. The exploration using Q-learning improves code coverage in comparison to random and model-based testing and is able to detect faults in applications under test.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Android, Test Input Generation, Reinforcement Learning, Q-Learning

ACM Reference Format:
Thi Anh Tuyet Vuong and Shingo Takada. 2018. 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 (A-TEST '18), November 5, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3278186.3278191

1 INTRODUCTION

The growing economy of mobile softwares at a global scale in recent years has led to a persistent demand for effective testing tools from developers. As a response to this demand, researchers have proposed different tools that automatically test mobile applications with as little human effort as possible. Most of the researchers choose Android platform to develop their tool because of the big market share of Android phone, Android testing tools would be undeniably useful and relevant to many developers. Besides, the open-source nature of the Android platform makes it easier for researchers to access both the application and the underlying operating system [11]. We target the Android platform and develop a testing tool for Android applications for the same reasons.

An Android application is composed of one or more *activities*, which are components in charge of the application's user interface (UI). Each activity contains various *UI components* such as button, text, check box, etc. An Android application is event-based, which means that its behavior is based on response to user's input events (*UI events*) such as click, scroll, text input, etc. It also reacts to *system events* from the phone's operating system such as phone call or SMS signal, etc. Because of the event-based nature of Android applications, a considerable number of researches has focused on automatically generating input events to conduct tests. Whatever the technique, the goal is to generate relevant inputs to reveal as many functionalities of the application as possible. However, researchers in this field still face two major challenges:

- Mobile applications usually contain a large set of components and possible events that can be performed. Therefore testing all possible combinations of components and events is time-consuming and difficult to scale to large systems. Automatic testing tools should test only the sequences of events which are relevant to the application, i.e., the sequences which lead to testing functionalities of the application.
- Some functionalities of the application can only be reached through a specific sequence of events (hard-to-reach functionality), making it difficult for automatic random testing to reveal and test them.

The main contribution of this paper is to propose an application of reinforcement learning, specifically Q-learning, to automatically generate test input for Android applications. Using reinforcement learning, we build a testing tool that interacts and explores the application's functionalities in a trial and error way. During this exploration, the tool dynamically builds a behavioral model of the application and generates test cases that follow the sequences of events which are the most likely to reveal application's functionalities. This paper also shows an evaluation of the tool in comparison with state of the art automated testing tools.

The paper is organized as follows: Section 2 reviews related works in Android testing techniques; section 3 provides an introduction to Q-learning then section 4 explains the proposed approach

A-TEST '18, November 5, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-6053-1/18/11…$15.00
https://doi.org/10.1145/3278186.3278191

and the implementation details of the Q-Learning tool; Section 5 discusses the evaluation of the proposed tool and finally section 6 concludes the paper along with suggestions for future works.

2 RELATED WORK

In this section, we investigate the existing techniques in automated test input generation for Android applications.

We can classify test input generation techniques based on exploration strategy [11]. The first technique is Random Exploration where the testing tool generates random events and sends them to the application. Although this technique is fast and simple, the generated events are usually insufficient. This is because the application can respond to only some of the many different types of events at each application state. Moreover, the events generated are usually redundant and have small chance to reveal hard-to-reach functionalities. Android Monkey [2] is a black-box testing tool that uses Random Testing strategy and it is widely used among developers because of its simplicity. It is particularly efficient in generating events (only UI events), capable of sending thousands of events per second to the application. Monkey usually obtains the highest code coverage among testing tools. However the crashes that it discovers are hard to reproduce and unrealistic from the viewpoint of human interaction with the application. It may be better to say that testing using Monkey is closer to stress test than functionality test. Dynodroid [18] is also a black-box testing tool that uses Random Exploration strategy but it has several improvements in comparison to Monkey. Instead of choosing a random event, it selects the event that is the most relevant to the current context of the application (BiasedRandom Strategy). Developers provide inputs which are impossible to be generated automatically such as authentication information to the tool before testing. Dynodroid also generates relevant system events by analyzing the listeners of the application. Another group of tools falling in this category are fuzzers: DroidFuzzer [29], Intent Fuzzer [25], Null Intent Fuzzer [5], etc. They mainly aim to generate invalid inputs that cause the application under test to crash.

The second technique is Model-based Exploration strategy where the testing tool builds a GUI model of the application under test, statically or dynamically, then uses this model to explore the application and generates events. A3E [10] proposes a dynamic depth-first exploration based on a dynamic model that considers each activity as a state of the application. This simplification of state representation leads to incompleteness of the test because A3E focuses on testing transitions between activities but not the behavior inside each activity. PUMA [14] is a dynamic analysis framework. It contains a generic UI automation capability (Monkey) that exposes high-level events for which users can define handlers. These handlers direct the Monkey's exploration, and also specify application instrumentation for collecting dynamic state information or for triggering changes in the environment during application execution. AndroidRipper [3] (also known as GUIRipper [8] or MobiGUITAR [7]) builds a model of the application by crawling. In each activity, it registers all the available events at the current state. Then it systematically executes all of them using Depth First Search strategy. AndroidRipper only generates UI events.

The last category of techniques is Systematic Exploration Strategy that uses algorithms such as symbolic execution and evolutionary algorithms to guide the exploration. This technique is used by tools such as EvoDroid [19] and Acteve [9]. EvoDroid proposes an evolutionary approach for system testing of Android applications. Evolutionary testing considers each test case as an individual. A population comprised of many individuals is evolved according to certain heuristics to maximize the code coverage and reduce the number of tests. EvoDroid statically analyzes the application under test to build its behavioral model. This model allows the evolutionary search to determine how the individuals should be crossed over to pass on their genetic makeup to future generations [19]. On the other hand, Acteve systematically generates test inputs based on concolic testing. It handles the path-explosion problem of concolic testing by proposing the notion of *subsumption* between event sequences to avoid redundant event executions.

While model-based strategy seems to be more efficient in comparison to random strategy for the same number of events generated, it still faces numerous challenges. Black-box model-based testing usually proceed by first building a model of the application statically before testing or dynamically during test by recording all the possible sequences of events from the running application. It then generates test cases to cover the sequences in the model [21] [30]. Therefore, the effectiveness of the generated test cases strongly depends on the completeness of the recorded model and how the application's states are represented. A widely used technique is to randomly traverse the GUI of the application under test to build a model, but this technique is redundant and limited when dealing with complex GUI. The reason is that some states are more reachable than the others so they will be executed more often under random strategy; whereas some hard-to-reach states may not be executed at all. A stopping condition is also hard to define, how much random exploration is enough to obtain a complete model?

Facing the problem of testing hard-to-reach GUI, Mariani et al. [20] proposed a black-box GUI testing tool for general Java desktop softwares that is capable of looking ahead and executing certain actions in order to get to a hard-to-reach GUI. They specifically use Q-learning, a reinforcement learning technique to achieve this. Their tool (called AutoBlackTest) builds a behavioral model dynamically and produces test cases incrementally, while interacting with the software under test. AutoBlackTest builds the behavioral model representation of the software under test as a multi-direction graph. Nodes represent states (GUIs), while edges represent executable actions. Connecting nodes corresponds to transition between states. At each step when the Q-learning algorithm explores the application and observes a new state or executes a new action, it extends the behavioral model. The performance is improved when the tool is provided with an initial test suite, which indicates how to test the most relevant functionalities of the software.

TESTAR [13] is a testing tool that also uses Q-learning to generate test sequences based on GUI information, without accessing the source code. A statistical analysis from the results of TESTAR points out that the Q-learning action selection strategy is only effective with an adequate choice of parameters. TESTAR can test general desktop softwares and web-based softwares.

Through AutoBlackTest and TESTAR, reinforcement learning technique has proved to be useful in GUI testing for standalone Java

applications. Such a tool, however, cannot be directly used to test Android applications because of the particularities of the Android platform as well as of Android applications themselves. Reinforcement learning needs to be adapted to the Android environment and need to show that it is useful in Android testing, hence the purpose of this research: An application of reinforcement learning in Android application testing.

The following section gives an introduction to reinforcement learning and Q-Learning, providing the necessary background knowledge to understand the rest of the paper.

3 Q-LEARNING

Q-Learning [26] is a well-known reinforcement learning technique, inspired by behaviorist psychology, where a software agent seeks to interact with an environment in a trial-and-error way. At each interaction, the agent performs an action, then based on the current state of the environment, it evaluates the consequences of that action in terms of its immediate reward and finally transit to a new state returned by the environment. The overall goal is for the agent to learn how to act in an optimal way that maximizes the cumulative reward, which is the reward collected when executing an entire sequence of actions.

The mathematical formulation of reinforcement learning problem is a Markov decision process. It is defined by [26]:

- S: Set of possible states
- A: Set of possible actions
- R: Distribution of reward given (state, action) pair
- P: Transition probability i.e distribution over the next state given (state, action) pair
- γ: Discount factor

Fig. 1 summarizes the reinforcement learning mechanism and the interactions between the environment and the agent.

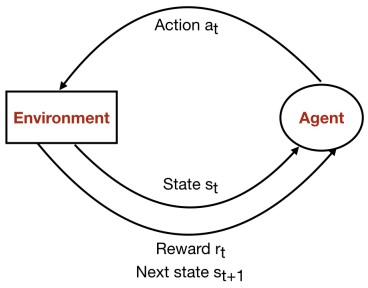

Figure 1: Reinforcement learning mechanism

A Q-learning agent interacts with the environment at each of a sequence of discrete time steps $t = 0, 1, 2, 3, \ldots$ [26]. At time $t = 0$, the environment starts from an initial state $s_0 \in S$. Then at each time step from $t = 0$ until done:

- The agent selects an action $a_t \in A(s_t)$ where $A(s_t)$ is the set of actions available in state s_t
- The environment samples and returns a numerical reward r_t based on the distribution of reward $R(.|s_t, a_t)$
- The environment samples and returns the next state s_{t+1} based on the transition probability $P(.|s_t, a_t)$
- The agent receives the reward r_t and finds itself in the new state s_{t+1}

At each step t, the agent observes the current state s_t of the environment and based on a policy π selects an action a_t. This policy represents the behavior of the agent toward the environment. Because we want the agent to behave in an optimal way that maximizes the cumulative reward, the goal of reinforcement learning is to find the policy π^* that maximizes the cumulative discounted reward $\sum_{t>0} \gamma^t r_t$ [17][26]

Given a policy π, in Q-learning, we define a function called Q-value function (or Q-function) that tells how good a (state, action) pair is. This function returns, for any combination of state s and action a, the expected cumulative reward that can be achieved by executing a sequence of actions that starts with a from s and then following the policy π. The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair, over all possible policies.

$$Q^{\pi}(s_t, a_t) = \max_{\pi} \sum_{t>0} (\gamma^t r_t | s = s_0, a = a_t, \pi) \quad (1)$$

If the optimal state-action values for the next step $Q^*(s_{t+1}, a_{t+1})$ are known, then the optimal strategy is to take the action that maximizes the expected reward of $r + \gamma Q^*(s_{t+1}, a_{t+1})$ where r is the immediate reward of the current step. Q^* satisfies the Bellman equation:

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \quad (2)$$

where γ is the discount-rate parameter between 0 and 1. It balances the relevance of immediate and cumulative reward: γ closer to 0 values more immediate reward and γ closer to 1 values more cumulative reward. The optimal policy π^* therefore corresponds to taking the action that has the highest Q-value as specified by Q^*

The Q-learning algorithm is based on equation (2) to estimate the value of the Q-function iteratively. The Q-function is initialized with a default value. Every time the agent executes an action a_t from state s_t to reach state s_{t+1} and receives a reward r_t, the Q-function is updated as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (3)$$

In this formula, α is the learning rate ($0 \leq \alpha \leq 1$). α represents the impact of a new observation on the estimated value of Q-function.

The Q-learning algorithm is guaranteed to converge to the true Q-function if applied to a Markovian environment, with a bounded immediate reward and with state-action pairs continually updated [28].

4 PROPOSED APPROACH: Q-LEARNING IN ANDROID APPLICATIONS TESTING

Because Q-Learning algorithm has the capability of finding the optimal way to reach a certain state of the environment, it can be used to guide the exploration to reach states that reveal application's functionalities or hard-to-reach states. In this paper, a Q-learning agent explores Android applications, which can be modeled as a collection of GUI states, and generate an event such as click, type, swipe, scroll, etc at each iteration. The agent learns gradually how to traverse the application in a way that reveals the application's

functionalities, then generate test cases that cover the most code possible.

In this paper, the Markov model is designed for the purpose of generating test input for Android application as follows:

- S: The set of application states. A state is defined by the activity name and the set of UI events available on the corresponding screen. The set of visited states is updated at each Q-learning iteration, by adding a new state each time one is visited.
- A: The set of possible GUI events. An event corresponds to an action that can be executed on a GUI component (for example, a button click). It is defined by a tuple (GUI component class name, event type, GUI component unique resource ID, position of the GUI component on the screen).
- R: The reward function that returns a numeric value given a transition (state, event, new state)
- P: The transition to a new state is determined by the response of the application under test when an event is executed at the current state
- γ: Discount factor

Our Q-learning testing tool consists of two main actors: The Environment and the Agent. The Environment represents the Android device and the application under test. It has three assistant modules: Observer, Executor and Reward Function. The Agent learns the most relevant Behavioral Model of the application under test and generates test cases based on this model. The Behavioral Model incorporates the result of Q-learning. It contains the state space S, the set of events that can be executed on each state and the estimated Q-function for each (state, event) pair.

In order to create a test case, the agent executes a sequence of a fixed number of events, also called an *episode*. After finishing an episode, the agent selects a random state from the ones that have already been visited and starts a new episode from there. The length of an episode is an important parameter affecting the effectiveness of the agent because it defines the size of the search space for each test case. This parameter depends strongly on the design of the application, such as the complexity of activities and how frequent the interaction between the different activities is. Each episode iteration proceeds as follow:

From t=0 until the end of the episode, repeat:

- STEP 1: **Environment - Observer** observes the application under test and builds the current abstract state.
- STEP 2: **Agent** acknowledges the current state and selects next event to execute based on a policy and its current behavioral model.
- STEP 3: **Environment - Executor** executes the selected event.
- STEP 4: **Environment - Observer** observes the new GUI state of the application
- STEP 5: **Environment - Reward function** calculates the consequent reward.
- STEP 6: **Agent** updates the Q-function.

Details of each of the steps are given below.

4.1 Observing the Environment (Step 1 and 4)

Observer is a module of the environment whose function is to observe the current state of the environment. The task of the Observer is to create an abstract presentation of the current GUI tree at each iteration, called shortly as a state. GUI tree is a hierarchy of GUI components that we can extract from the application under test during runtime using Android UI Automator. An abstract GUI state consists of the current activity's name and a tuple of all the UI events in the current GUI tree. The observer only includes in the abstract states the GUI elements belonging to the current application under test (by checking which package the GUI element belongs to). In this paper, only event types of click, long-click, check (check box), text input and scroll are considered. Even though the Android platform supports a larger set of event types, we currently support only these five as they are the most widely used, and consider others as part of future work. Other than UI events of the application under test, the observer takes into account the *menu* button click and the *back* button click of the device as available UI events.

4.2 Selecting an Event to Execute at Current State (Step 2)

At each time step, the agent selects an event to execute given the current state. This event is chosen based on a policy π. The challenge of choosing a good policy is to balance the trade-off between *exploration* and *exploitation* [26]. In order to obtain a big reward, the agent must prefer the events that it has tried in the past and found to be effective in producing reward. It should exploit what it knows about the application and selects the event with the highest Q-value. But to select such event, the agent also has to explore new events in order to make better selections in the future. Therefore, probabilistic approaches are commonly considered. In this research we use a popular policy called ϵ-greedy. The agent selects either a random event among the ones available in the current state, with probability ϵ, or the event with the highest Q-value according to the current behavioral model, with probability $1 - \epsilon$.

At the beginning of the testing process, we want the agent to explore as many states as possible in order to build the closest to complete model of the application, therefore, a big value of ϵ should be used. However, once the model is pretty much built, we want the agent to follow the Q-value function to quickly reach functionalities of the application and test them, therefore a smaller value of ϵ is expected. In this paper, we start with $\epsilon = 1$ to enable maximum *exploration* then we decrease its value uniformly during the first 100 episodes until a final minimum value (which is fixed at 0.5 after empirical study during evaluation) to change the behavior of the agent toward *exploitation*.

4.3 Executing an Event in the Environment (Step 3)

The Executor module executes an event on an actual Android device or an emulator through UI Automator [15] and Android Debug Bridge [1]. The Executor can execute all the event types that are observable by the observer: click, long-click, check, text input, scroll. For text input, the text are generated randomly during execution.

4.4 Calculating Reward (Step 5)

The reward is calculated in the environment by a reward function R, which basically tells the agent which event is good and which one is bad. Two aspects are taken under consideration when defining the reward function: GUI change and execution frequency.

To heuristically identify the events that trigger new functionalities, the reward function favors the events that lead to many changes in the abstract GUI state and give them a higher reward. This reasoning is similar to the one used in AutoBlackTest [20]. Given two states s_1 and s_2, the reward function calculates the degree of change from s_1 to s_2 by comparing and identifying the number of GUI events in s_2 but not in s_1, described as $|s_2 \backslash s_1|$. The relative change is then defined by the ratio $|s_2 \backslash s_1|/|s_2|$ where $|s_2|$ is the number of GUI events in $|s_2|$. This formula considers the components that newly appear in s_2 but not the components that disappear from s_1. This avoids giving too large a value to events that trigger jumps between application's activities. It moderately increases the Q-values, therefore we can test both transitions between activities and functionalities inside each activity.

Another factor is added to help the agent balance the exploration and exploitation: execution frequency. We count the number of times each event has been executed, and decrease the reward as the execution frequency increases. This reward value becomes relatively small in comparison to the GUI change reward after a certain number of iterations. But it helps the Q-agent to explore new states faster at the beginning of the test, hence improving the completeness of its state space.

In summary, the definition of the reward function is given by the formula:

$$r_t = R(s_t, a_t, s_{t+1}) = \frac{|s_{t+1} \backslash s_t|}{|s_{t+1}|} + \frac{1}{f(s_t, a_t)} \quad (4)$$

where $f(s_t, a_t)$ is the execution frequency of event a_t in state s_t up until time t.

4.5 Updating the Q-Value Function (Step 6)

The agent builds and updates the Q-value function, i.e the behavioral model according to the transition that occurs during the current iteration.

The agent updates the Q-value function after each iteration using equation (3), with the learning rate $\alpha = 1$ and the discount-rate $\gamma = 0.9$. A learning rate α closer to 1 imposes a bigger impact of new observations on the model. We choose the value $\alpha = 1$ so that the agent learns quickly how the application behaves. The discount factor balances the relevance of the immediate reward with respect to future events, and the value 0.9 maximizes the reward collected during an entire episode, rather than maximizing the immediate reward. Equation (3) thus becomes:

$$Q(s_t, a_t) \leftarrow r_t + 0.9 \max_a Q(s_{t+1}, a) \quad (5)$$

When an action reaches a state that hasn't been visited, the second term of the formula is zero as we don't have any estimation of the Q-value for that state. In that case, the Q-value is simply the immediate reward. Otherwise, the Q-value is computed according to both the reward value and the Q-values of the events in the target state.

A special function is used to handle cases where the transition leads to exiting the application under test. In that case, the testing tool randomly executes up to a maximum number of events which don't belong to the application under test. This maximum number is fixed at two events. After each execution, we check if the tool goes back to the application under test. If yes, we proceed the test as normal; if not, we force the tool to go back to the application's launcher activity. The event that triggers exiting the application is given 0 reward, as we avoid executing it. This is necessary since we would not be able to differentiate between a normal application exit and an application crash (which we consider as the existence of a fault).

4.6 Implementation

The tool is written with Python 2.7. The architecture and workflow of the tool is shown in Fig. 2. In order to interact with the Android device and with the application under test, we use a python library for Android UI Automator [15] along with Android Debug Bridge [1].

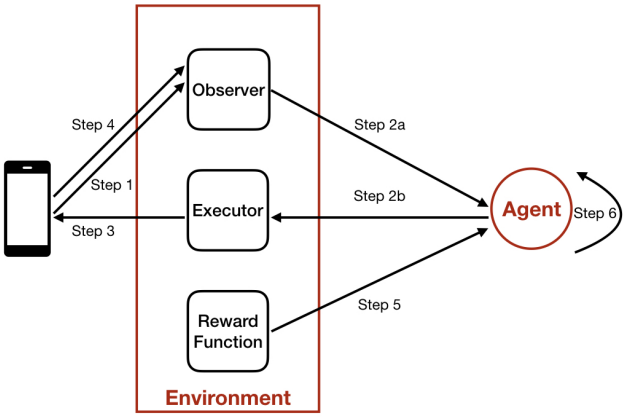

Figure 2: Architecture of the Q-Learning tool

5 EVALUATION

5.1 Metrics of the Evaluation

The metric for measuring the effectiveness of the tool is code coverage, a common metric for evaluating testing tools. Specifically, we use line coverage to evaluate the Q-Learning tool and to compare with other testing tools because it is the smallest unit of coverage.

The evaluation of the tool aims to answer the following research questions:

- RQ1: Does the tool improve code coverage in comparison to state-of-the-art testing tools?
- RQ2: Is the tool capable of detecting faults?

The tool is compared with state of the art testing tools: Android Monkey, Dynodroid and PUMA.

Table 1: Target applications for evaluation

Application name	Version	Lines of code	Category
AnyMemo	8.3.1	8335	Education
Battery Dog	0.1.1	463	Tools
Learn Music Note	1.0.2	399	Puzzle
Munch Life	1.4.2	161	Entertainment
My Expenses	1.6.0	2842	Finance
Tippy Tipper	1.1.3	990	Finance
Who has my stuff	1.0.7	636	Productivity

5.2 Target Applications

We evaluated our tool on seven open-source Android applications: AnyMemo [22], Battery Dog [16], Learn Music Notes [24], Munch Life [23], My Expenses [27], Tippy Tipper [12] and Who has my stuffs [6]. They are of various size and complexity, with both static and dynamic content. Table 1 provides a short description of each application.

5.3 Evaluation Setup

Each tool is run on a separate virtual machine provided by AndroTest [11]. AndroTest provides a good benchmark for evaluating different testing tools on a collection of various applications. All virtual machines run Ubuntu 32-bit with 6114 MB of base memory and 2 processors. For each testing session, an application was installed on an emulator with only default setting. To prevent parasitic data from previous sessions, all the data on the emulator was deleted and the emulator was restarted before starting a new test. Each testing tool was run on each application four times, then an average is taken as the final result.

5.4 Obtaining Results

5.4.1 ***Code Coverage.*** Code coverage is obtained using Emma [4], a helper class which is embedded in the application under test before running the test. A custom script is made to pull the code coverage file from the device and convert it to a human-readable report every five minutes. The report provides details about package, class, block and line coverage. The coverage returned by Emma takes into account the lines of code of the instrumentation classes. We calculate the real coverage (which counts only the lines of code of the application) based on the results by Emma and use it in evaluation.

5.4.2 ***Application Faults.*** Application faults are discovered while running the tool, when the application crashes unexpectedly. The stack traces of the errors are recorded by logcat, a command-line tool that dumps system messages. Error traces are retrieved from the phone or emulator through Android Debug Bridge (adb).

5.5 Result of the Evaluation

The code coverage obtained by our tool is compared with the results for Android Monkey, Dynodroid and Puma. We execute each tool for one hour, at which time we stop it, i.e. we compare the results after executing for one hour.

Table 2: Code coverage after one hour (in %)

App	Lines	Qtool	Dyno	Puma	Mnk
AnyMemo	8335	44.19	21.03	fail	25.39
Battery Dog	463	65.37	66.63	11.8	72.33
Learn Music Note	990	63.08	62.13	34.40	63.35
Munch Life	161	87.39	70.82	50.8	91.91
My Expenses	2842	53.62	23.74	38.20	55.80
Tippy Tipper	990	85.44	49.68	85.44	81.20
Who has my stuff	636	65.27	61.83	58.5	79.11

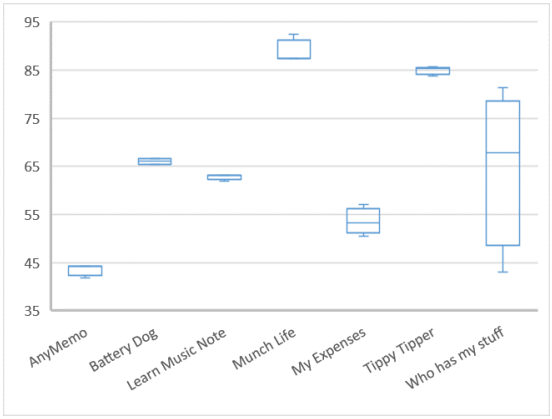

Figure 3: Variance of code coverage (%) achieved by the Q-Learning tool

5.5.1 ***RQ1: Does the Tool Improve Code Coverage in Comparison to State-of-the-Art Testing Tools?*** Table 2 compares the code coverage of the seven target applications and Fig. 3 shows the variance of code coverage achieved by the Q-learning tool.

The proposed approach outperforms DynoDroid and Puma for 6 out of 7 applications, except for Battery Dog where our tool obtains the code coverage slightly less than Dynodroid. This difference can be explained by the fact that Dynodroid can generate system events and UI events while our tool can only generate UI events. In the case of Battery Dog, the application monitors the battery level of the phone and it responds to system events, which our tool cannot handle.

For most of the applications, Android Monkey obtained a higher code coverage because it is able to effectively generate a large amount of events per second, much more than our tool.

5.5.2 ***RQ2: Is the Tool Able to Detect Faults in the Application?*** Even though code coverage is a good indicator for the effectiveness of a testing tool, the ultimate goal of testing is to discover faults in applications. Therefore we need to evaluate the tool in term of faults detected. During each test, log from the device is recorded though Android Debug Bridge and then analyzed to discover faults related to the application. For each application, we count the number of errors (or crashes) caused by the application during test and the number of distinct faults. Table 3 summarizes the results of the experiment. Our tool was able to trigger crashes and

Table 3: Number of crashes and distinct faults discovered by the Q-learning tool during test

App	Lines of code	Crashes	Distinct faults
Any Memo	8335	215	14
Battery Dog	463	5	3
Learn Music Note	990	40	1
Munch Life	161	0	0
My Expenses	2842	20	1
Tippy Tipper	990	0	0
Who has my stuff	636	16	1

discover faults in most of the applications. In several applications, multiple crashes were caused by the same fault.

6 THREATS TO VALIDITY

The two main threats to validity are internal and external validity. One threat to external validity is the number of applications used for evaluation. Although we only had seven applications, we tried to minimize this threat by choosing applications from different categories and different sizes.

A threat to internal validity is the non-determinism of our approach, which could result in obtaining different code coverage for each run. Thus we executed multiple runs to reduce this threat.

7 CONCLUSIONS

This paper has proposed a reinforcement learning approach to Android application testing, using Q-learning algorithm. The proposed tool is based on a Q-learning agent and build the behavioral model of the application gradually by letting the agent interact with the application in a trial-and-error way. The Q-learning agent explores the application in an optimal way, reaches the most relevant functionalities of the application, resulting in the generation of test cases. Evaluation of the tool has proven that the tool offers significant improvement in code coverage in comparison to existing testing tools, and it is capable of discovering faults.

Future works to improve our tool include a detailed investigation into the reward function for the Q-learning algorithm, supporting more types of events such as system and context events, and handling nondeterminism.

REFERENCES

[1] Android Debug Bridge. Retrieved June 4, 2018 from https://developer.android.com/studio/command-line/adb

[2] Android Monkey. Retrieved June 4, 2018 from https://developer.android.com/studio/test/monkey

[3] Android Ripper. Retrieved June 4, 2018 from https://github.com/reverse-unina/AndroidRipper

[4] EMMA: a free Java code coverage tool. Retrieved June 4, 2018 from http://emma.sourceforge.net

[5] Intent Fuzzer. Retrieved June 4, 2018 from https://www.nccgroup.trust/us/our-research/intent-fuzzer/

[6] Who has my stuffs. Retrieved June 4, 2018 from https://f-droid.org/en/packages/de.freewarepoint.whohasmystuff/

[7] D. Amalfitano, A.R. Fasolino, P. Tramontana, B.D. Ta, and A.M. Memon. 2015. MobiGUITAR – a tool for automated model-based testing of mobile apps. *IEEE Software* 32, 5 (Oct. 2015), 53–59. https://doi.org/10.1109/MS.2014.55

[8] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. 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. https://doi.org/10.1145/2351676.2351717

[9] S. Anand, M. Naik, M. J. Harrold, and H. Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)*. ACM, New York, NY, USA, Article no. 59. https://doi.org/10.1145/2393596.2393666

[10] Tanzirul Azim and Iulian Neamtiu. 2013. A3E - Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications (OOPSLA 2013)*. ACM, New York, NY, USA, 641–660. https://doi.org/10.1145/2491411.2491450

[11] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 429–440. https://doi.org/10.1109/ASE.2015.89

[12] Bryan Denny. Tippy Tipper. Retrieved June 4, 2018 from https://github.com/mandlar/tippytipper

[13] Anna I. Esparcia-Alcazar, Francisco Almenar, Urko Rueda Mirella Martinez, and Tanja E.J. Vos. 2016. Q-learning strategies for action selection in the TESTAR automated testing tool. In *Proceedings of META 2016 6th International Conference on Metaheuristics and Nature Inspired Computing (META 2016)*. 174–180.

[14] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. 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 (MobiSys '14)*. ACM, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390

[15] Xiaocong He, Yuanyuan Zou, Qian Jin, Xu Jingjie, and Xia Mingyuan. UI Automator Library for Python. Retrieved June 4, 2018 from https://github.com/xiaocong/uiautomator

[16] Ferenc Hechler, Gary Oberbrunner, and Greg Willard. AndroidBatteryDog. Retrieved June 4, 2018 from https://github.com/fdroid-revivals/AndroidBatteryDog

[17] Fei-Fei Li, Justin Johnson, and Serena Yeung. Stanford course CS231n: Convolutional Neural Networks for Visual Recognition, 2017, Lecture 14. Reinforcement Learning. Retrieved June 3, 2018 from http://cs231n.stanford.edu/

[18] 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. https://doi.org/10.1145/2491411.2491450

[19] R. Mahmood, N. Mirzaei, and S. Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 599–609. https://doi.org/10.1145/2635868.2635896

[20] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. 2014. Automatic testing of GUI-based applications. *Software Testing, Verification and Reliability* 24, 5 (Aug. 2014), 341–366. https://doi.org/10.1002/stvr.1538

[21] A.M. Memon, M.L Soffa, and M.E Pollack. 2001. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE)*. ACM, New York, NY, USA, 256–267. https://doi.org/10.1145/503209.503244

[22] H. Ning. AnyMemo. Retrieved June 4, 2018 from https://f-droid.org/en/packages/org.liberty.android.fantastischmemo/

[23] Blaine Pace. Munch Life. Retrieved June 4, 2018 from https://github.com/pacebl/MunchLife

[24] F.C Puig. LearnMusicNotes. Retrieved June 4, 2018 from https://github.com/FerCa/LearnMusicNotes

[25] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis and Software and System Performance Testing, Debugging, and Analytics (WODA+PERTEA 2014)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/2632168.2632169

[26] Richard S. Sutton and Andrew G. Barto (Eds.). 1998. *Reinforcement Learning An Introduction*. MIT Press, Cambridge, MA.

[27] Michael Totschnig. MyExpenses. Retrieved June 4, 2018 from https://github.com/mtotschnig/MyExpenses

[28] Christopher J.C.H. Watkins and Peter Dayan. 1992. Technical Note: Q-Learning. *Machine Learning* 8, 3-4 (May 1992), 279–292. https://doi.org/10.1023/A:1022676722315

[29] H. Ye, S. Cheng, L. Zhang, and F. Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing and Multimedia (MoMM)*. ACM, New York, NY, USA, 68. https://doi.org/10.1145/2536853.2536881

[30] X. Yuan, M.B Cohen, and A.M Memon. 2011. GUI interaction testing: incorporating event context. *IEEE Transactions on Software Engineering* 37, 4 (July 2011), 559–574. https://doi.org/10.1109/TSE.2010.50