Applying Reinforcement Learning for Automated Testing of Mobile Application Focusing on State Definition, Reward, and Learning Method

Keita Murase
Dept. of Info. and Comp. Sci., Keio University
murase42@doi.ics.keio.ac.jp

Shingo Takada
Dept. of Info. and Comp. Sci., Keio University
michigan@ics.keio.ac.jp

Abstract

There have been various studies on the automation of mobile app testing. Typical methods for automated testing of mobile apps are based on random search and on building state transition models. But there are problems in terms of the efficiency of search and accuracy of model building. This paper focuses on applying reinforcement learning to testing of mobile apps, especially issues such as explosion of the number of states, fixed rewards for transitions, and difficulty in convergence of learning. We focus on state definition, reward function, and a learning method to solve these problems. Specifically, we define states using discrete values of UI (User Interface) information on the screen, define a dynamic reward function, and perform periodic learning by using the transition history. The proposed method is implemented and evaluated. Evaluation results show that our proposed approach shows 1.21 times higher coverage than an existing tool using reinforcement learning.

1. INTRODUCTION

Mobile apps are applications that run on smartphones and other devices, and play a very important role in our daily life. Among smartphone OSs, Android has the largest market share. Android apps need to be tested just like any other software, such as through GUI testing. GUI testing of Android apps involves actually performing operations on the screen displayed on the device and checking for anomalies. However, manual GUI testing is very expensive when the scale of the application is large or the frequency of updates is high. Automated GUI testing will reduce this cost.

Several methods have been proposed for Android GUI testing. One of the most well-known methods of GUI test automation is random testing. Random testing randomly selects and executes operations on the screen, and is used in the well-known tool Monkey [9]. However, since the selection of operations is random, there are problems with the efficiency and stability of the search. For improving the search efficiency, model-based methods were proposed [12][4]. In the model-based method, the available actions and the states that can be reached by the actions in each state are obtained in advance by static analysis of the code and are represented in the form of a state transition graph enabling efficient path searches. However, accurate analysis of Android apps is difficult, and it is often impossible to construct state transition diagrams correctly.

Reinforcement learning is a well-known method for automatic game playing, and recently, several methods have been proposed using reinforcement learning for automatic testing of Android [6][7][8]. In reinforcement learning, rewards are given for state transitions, and search strategies are learned so that the rewards obtained are large. The advantage of this method is that it does not depend on the accuracy of the model since the strategy is based on the actual state transitions.

However, most approaches give the same reward to the same transition regardless of the situation, which may lead to the same transition to be always chosen. Thus, we propose a method to improve the coverage of the test by giving rewards for reinforcement learning according to the search situation. Naive application of this may lead to difficulty for the learning results to converge. Thus, we propose a method to cope with this convergence problem by storing the history of transitions, and periodically repeating the learning focusing on the most recent transitions.

This paper is organized as follows: Section 2 provides a brief introduction to reinforcement learning. Section 3 reviews related work. Section 4 describes our proposed approach and implementation. Section 5 evaluates our approach and section 6 makes concluding remarks.

2. REINFORCEMENT LEARNING

Reinforcement learning proceeds through the interaction between the environment and the agent. The basic flow is as follows:

DOI reference number: 10.18293/SEKE23-154

1. The agent performs an action based on a strategy.
2. Based on the agent's action, the environment changes its state.
3. The environment rewards the agent as a result of the action.
4. The agent improves its strategy based on the reward.

As we will show in the next section, reinforcement learning has been applied to GUI testing of Android apps, where environment is the Android device, an action is an operation on the Android device, and state is a state of the screen.

Q-learning [10] is a popular reinforcement learning algorithm. It represents a strategy in terms of a Q-function, and is an algorithm for learning a Q-function. The Q-function $Q(s, a)$ is the value of the action a in the state s. If this value is correct, the reward can be increased by choosing an action with a large Q-function in each state. If an action a_t in state s_t results in a transition to state s_{t+1} and a reward r_t, the value of $Q(s_t, a_t)$ is estimated by Q-learning as in Formula (1). $\gamma (0 \leq \gamma \leq 1)$ is the discount rate, which indicates how much of the future reward is taken into account.

$$Q(s_t, a_t) = r_t + \gamma \max Q(s_{t+1}, a_{t+1}) \tag{1}$$

With the learning rate $\alpha (0 \leq \alpha \leq 1)$, which indicates how much the Q-function is changed, the value of the Q function can be updated as in Formula (2).

$$\begin{aligned} Q(s_t, a_t) = {} & Q(s_t, a_t) \\ & + \alpha(r_t + \gamma \max Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \end{aligned} \tag{2}$$

3. RELATED WORKS

Monkey [9] is one of the most widely used tools for automated testing of Android apps. The tool randomly generates events such as click operations. However, since the selection of actions is random, there are problems in terms of efficiency and stability of the search. In this study, we aim to improve the efficiency of search by using reinforcement learning.

Some studies have attempted to construct a state transition model and search on it. Yang et al.[12] proposed Window Transition Graph, which supports multiple windows and system events, and performed path exploration and test case generation on the graph. Lai [4] et al. proposed Screen Transition Graph, which represents the state transitions of the Android screen by static analysis, considering background execution and screen composition. They have also developed Goal Explorer, which automatically and efficiently tests specific functions by prioritizing the direction near a specific function on the graph. The problems with these methods are that they are prone to omissions of states and transitions when analyzing and modeling Android apps and that they are unable to deal with cases in which transitions include randomness. When reinforcement learning is used, efficient search can be performed by taking probabilities into account based on the actual execution results.

There are also approaches that focus on heuristics. TimeMachine [2] saves various states that are expected to be visited as "interesting states" and takes a snapshot of the machine state so that it can be resumed from that state at any time. Then, when the machine is in a situation where it only visits states that it has already visited, it can resume from the interesting state to facilitate the search.

Recent testing work has applied reinforcement learning for testing Android apps [8][7][6]. Qdroid [8] is based on Deep Q-Network. It groups GUI components according to their semantics and decides which group of GUI components to act on. The reward is larger when the screen changes, making it easier to avoid meaningless actions that have no effect on the screen. However, if the reward does not change with time, the actions taken in each state will become constant when the learning process is completed. This results in the Q-function to converge to a constant value, causing the same transition to be repeated. This is contrary to the purpose of the test, which is to perform a wide variety of transitions. ARES [7], like Qdroid, gives a large reward when a screen change occurs, and also gives a reward when a bug is found. Similar to Qdroid, it is highly likely that the same transitions that eventually yield high rewards are repeated after the learning process converges. Q-testing [6], on the other hand, introduces rewards that change with time. It uses Siamese Network to judge whether the screen is similar to a screen already visited, and gives higher rewards when a similar screen has not been visited. However, when the reward changes over time, the problem arises that learning may not converge.

4. PROPOSED APPROACH: IMPROVING REINFORCEMENT LEARNING ALGORITHM FOR EXPLORATION

Based on the problems in the previous studies, we focus on three aspects of applying reinforcement learning, and propose a definition for each of them. First, we focus on state definitions, and characterize the screen in such a way that the number of states is reduced by using information from UIAutomator [11]. Second, we focus on the definition of rewards, and define it so that it changes according to the current search status. Finally, we focus on the learning process, and make it iterative so that the reward converges even if the reward is dynamic. We describe each of these in more detail in the rest of this section.

4.1. State and Action

The state definition is based on the following attributes, which are obtained from UIAutomator [11].

- Resource ID
- Possible operations: clickable, long clickable, scrollable, checkable, focusable

Resource ID is a feature assigned to each UI element. For example, Figure 1 is the initial screen of the card game `Hot Death` [3]. The NEW GAME button has the Resource ID *com.smorgasbork.hotdeath:id btn_new_game*. In the screen shown in Figure 1, all five buttons are assigned different Resource IDs, but there are cases where multiple UI elements have the same Resource ID. This will be handled using actions, described later in this subsection.

UIAutomator also provides information about possible operations, such as whether each UI element is clickable or not, and whether it is long-clickable or not.

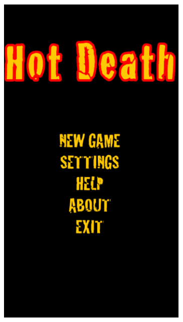

Figure 1. Screen 1

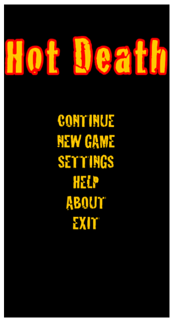

Figure 2. Screen 2

Based on information obtained by UIAutomator, we group the items whose Resource IDs and possible operations are the same, and define the state using the size of each group. For example, in Figure 1, if we assign five buttons to the first five components, the state is defined as the vector $(0, 1, 1, 1, 1, 1)$. Note the "0" in the first element. UI elements that are not currently on the screen are also included in the state definition where the corresponding element size is 0, Thus, Figure 1 has an element that is not visible. In Figure 2, we can now see the CONTINUE button, which makes the vector to be $(1, 1, 1, 1, 1, 1)$. The size that is necessary for a screen depends on the app, but Hot Death requires about 100-dimensional vectors. Coordinates of each UI element can also be obtained with UIAutomator, but since they have continuous values, we have omitted them to avoid an explosion of the number of states.

For a given UI element, one of the possible operations is chosen as an action. At this point, an action value is defined for the pair of state and that UI element. There may be a case where there are multiple UI elements with the same Resource ID. Although we do not save the coordinates of the UI elements, the order of the UI elements is based on the coordinates. This enables us to calculate the action value even if there are multiple UI elements with the same Resource ID. Note that we can also check if the action is a valid operation for the the corresponding UI element. For example, a click is always applied to a button for which only the clickable attribute is true, and a random string is an input to a UI element of the EditText class.

4.2. Reward

First, we define the penalty $penalty(s_t, a_t)$ for performing action a_t in state s_t as in the following Formula (3).

$$penalty(s_t, a_t) = \sum_s \frac{count(s, a_t)}{distance(s_t, s) + 1} \tag{3}$$

where $count(s, a_t)$ is the number of times an action a_t has been performed in state s in the past, and $distance(s_t, s)$ is the Manhattan distance between the vectors of state s_t and state s. In other words, the more times the same action has been performed and the more similar s_t is to the state in which the action was taken, the larger penalty is given.

With the penalty, we define the reward $r(s_t, a_t)$ for action a_t in state s_t as in the following Formula (4).

$$r(s_t, a_t) = \begin{cases} X_{high} & (penalty(s_t, a_t) < P) \\ \frac{X_{low}}{penalty(s_t, a_t)} & (penalty(s_t, a_t) \geq P) \end{cases} \tag{4}$$

If the penalty is less than the threshold P, a large constant reward X_{high} is given, and if the penalty is above the threshold, the small reward reduced according to the penalty is given. The reason for the division according to the threshold is to emphasize the importance of the first transition which has a large significance in the test. In the above reward definition, the reward changes according to the situation of the search. It leads to selecting actions according to the situation.

In Figure 1, consider the penalty for selecting the HELP button. Since the distance from the state of Figure 1 is naturally 0, if the HELP button has been selected three times in the past, $\frac{3}{1}$ is added to the penalty. Since the distance from the state of Figure 2 is $distance((0, 1, 1, 1, 1, 1), (1, 1, 1, 1, 1, 1)) = 1$, if the HELP button was selected three times in the past in Figure 2, the penalty is $\frac{3}{2}$. If the HELP button is selected three times in each of the two screens, the penalty is 4.5, and the reward is $\frac{X_{low}}{4.5}$, if this is greater than or equal to the threshold value P.

4.3. Learning

Q-function can be kept in a tabular form with each row as a state and each column as an action. The value of each cell represents the value of an action in a certain state. When a new state is visited, a new row is added and the reward is initialized based on Formula (4). Since it is difficult for the Q-function to converge by simply learning based on the update formula of the Q-learning at each state transition, the learning is periodically iterated as follows:

1. Recalculate the reward for the previous transitions.
2. Take the last N transitions in the order of newest to oldest and repeat updating the Q-function.
3. Randomly select the previous transitions, and repeat updating the Q-function.

First, we recalculate the transitions that have been executed, since it is highly likely that the rewards have changed since the transition occurred. Next, we iteratively train with the most recent transitions that may not have been reflected yet. In this iteration, the transitions are taken from the most recent ones so that the most recent results can be efficiently propagated to the traversed routes. Finally, we aim to bring the overall Q-function close to the correct value by repeating the learning process with a random selection of all the transitions up to now.

4.4. Implementation

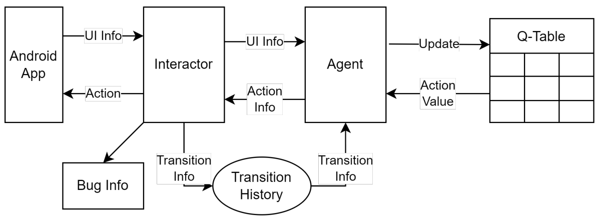

Figure 3. Tool Architecture

We implemented our tool based on Qdroid [8] (Figure 3). Interactor interacts with the Android app by using UIAutomator [11]. Interactor separates the screen data in XML format received from UIAutomator into an array of UI elements and sends the UI information to the Agent. Menu and Back buttons are added as virtual elements to the array even if they do not appear on the screen, so that they can be handled if they are operable. We also add randomly positioned buttons to the array. These buttons are added as UIAutomator may not recognize some UI elements.

The Agent obtains UI information from Interactor, selects the next action, and updates the Q-table. When selecting the next action, Agent selects the action which has the largest Q-function with high probability, and selects the action randomly with low probability, based on ϵ-Greedy. The selected action information is sent to Interactor, which then performs that action through UIAutomator. Additionally, Interactor collects stack trace of crashes with logcat [5].

5. EVALUATION

We consider the following three research questions for evaluation:

- RQ1: How is the performance compared to other tools?
- RQ2: How much influence do our changes have?
- RQ3: How does coverage change over time?

Evaluation was performed on the virtual environment provided by Androtest [1] in the following execution environment:

- OS: Ubuntu 14.04.1 LTS
- CPU: AMD Ryzen Threadripper 3990X 64-Core
- Memory: 6113MB
- Emulator: Android 4.4

The target apps are the same as Qdroid. We measured the average method coverage and the average number of unique crashes over three trials for each app.

5.1. RQ1: How is the performance compared to other tools?

We targeted Qdroid, ARES and Q-testing for our evaluation. Unfortunately, although the Q-testing executable was made available[1], we could not make it execute in our environment. A similar situation occurred with ARES[2], where we could make it execute for some apps, but not the ones we were targeting. Thus, we focus on Qdroid for comparison.

Table 1 shows the results of the coverage and the number of crashes between Qdroid and our proposed approach ("Our tool"). We will discuss Qdroid2 in RQ2.

Both coverage and number of crashes showed the same trend. First, the coverages of `Anymemo`, `Multi SMS Sender`, `MunchLife`, and `Weight Chart` were improved. This is considered to be due to the increase of search space by the changes in the UI recognition method, including the addition of virtual UI elements. For example, in `MunchLife`, menu button

[1]https://github.com/anlalalu/Q-testing
[2]https://github.com/H2SO4T/ARES

Table 1. Evaluation results

Application	Coverage (%)			# Crashes	
	Qdroid	Qdroid2	Our Tool	Qdroid	Our Tool
Any Memo	43.2	47.1	49.3	3.7	6.0
Dalvik Explorer	82.6	80.5	83.7	0	0
Hot Death	64.8	79.0	80.2	0	0.7
Mileage	38.2	43.2	50.4	0.7	1.7
Mini Note Viewer	59.6	51.4	48.7	1	0.7
Multi SMS Sender	37.9	64.7	66.9	0	0
Munch Life	53.8	92.3	92.3	0	0
My Expenses	62.5	60.4	62.3	0	0
Random Music Player	58.7	58.7	58.7	0	0
Tippy Tipper	56.1	82.8	88.1	0	0
Weight Chart	46.1	64.2	73.6	0	0
Who has my stuff	89.1	75.6	83.7	0	0
Average	57.7	66.7	69.8	5.4	9.1

does not appear as UI on the screen even when the screen transition by menu button is possible. In our tool, the menu button is added as a virtual UI element, so the tool can reach a state that can only be reached from the menu. In addition, although there is no corresponding UI element in `Weight Chart`, there is a situation in which touching any part of the screen causes a screen transition. The proposed tool can handle this situation by adding random buttons.

Reinforcement learning worked well in the cases of `Hot Death`, `Mileage`, and `Tippy Tipper`. For example, in the case of `Tippy Tipper`, the user exits from the final destination screen by pressing the Back button and restarting the program. Unless the Back button is used properly, the efficiency of the program will be reduced.

The coverage was slightly lower for `Who has my stuff` and `Mini Note Viewer`. This is because we added a virtual menu button regardless of its existence as a UI element. Thus this addition was superfluous.

5.2. RQ2: How much influence do our changes have?

There are two major differences between our approach and Qdroid: the method of acquiring UI information (adding virtual UI elements) and the reinforcement learning algorithm (state definition, reward function, and periodic learning). In order to measure the influence of the change in the reinforcement learning algorithm, we implemented the part of Qdroid related to the UI acquisition and the operation of the app in the same way as the proposed method and measured the coverage. We call this tool "Qdroid2". The results are shown in Table 1.

The results show that the change of the UI acquisition method (i.e., Qdroid2) improved the coverage of Qdroid from 57.7% to 66.7%, and the change of the reinforcement learning algorithm further improved the coverage to 69.8%. We believe that our UI acquisition method led to a widening of the search area leading to the increase in coverage.

However, some apps showed a decrease in coverage. For example, the coverage dropped from Qdroid to Qdroid2 in `Dalvik Explorer`, `Mini Note Viewer`, `My Expenses`, and `Who has my stuff`. This suggests that the addition of virtual UI elements will lead to an increase in the number of extra action options, which may be unnecessary. Still, except for `Mini Note Viewer`, the coverage for our tool increased from Qdroid2 to at least nearly the same result as Qdroid due to changes in the reinforcement learning algorithm, most likely due to avoiding unnecessary actions.

5.3. RQ3 : What is the change of coverage over time?

Figures 4 and 5 show how coverage changed over a two hour period for `Munch Life` and `Mileage`, respectively. The coverage was measured every five minutes, and the results are the average of three trials. Including other apps which are not shown (due to space), most of the apps show a rapid increase in coverage immediately after starting, followed by a gradual increase in coverage. This makes sense as many codes are involved in the process immediately after starting.

The coverage in some apps, such as `Munch Life` (Figure 4), reached its maximum value at an early stage. This is likely because the app was small and thus the searchable portions were quickly exhausted. Another app that showed this same trend was `Random Music Player`, where a particular type of input, in this case appropriate URLs, were necessary. Our tool cannot automatically generate such URLs, but this is an area of future work.

The coverage in other apps, such as `Mileage` (Figure 5), was still increasing after two hours, especially for our tool. These apps were large or require complicated procedures to execute their functions. For example, `Mileage` has a wide variety of settings for functions related to the mileage of a car, such as the type of car, the unit of distance, etc. `Any Memo` and `My Expenses` were two other examples of large app showing an increase after two hours. On the other hand, `Hot Death` also was fairly large, but it did not increase as much, most likely due to code related to winning which is difficult to reach with automatic execution.

5.4. Threats to Validity

One threat to validity is the number of apps tested which was 12. We chose these 12 as they were used in other research. But since even our results showed that coverage varies greatly depending on the app, we consider further evaluation as future work.

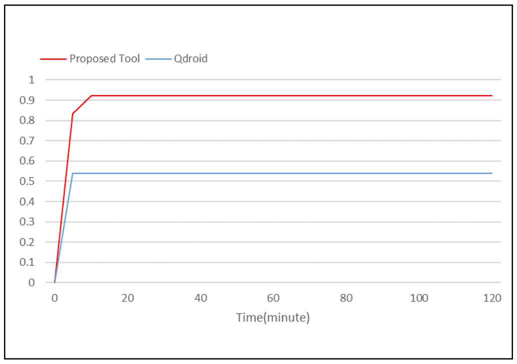

Figure 4. Coverage Evolution in Munch Life

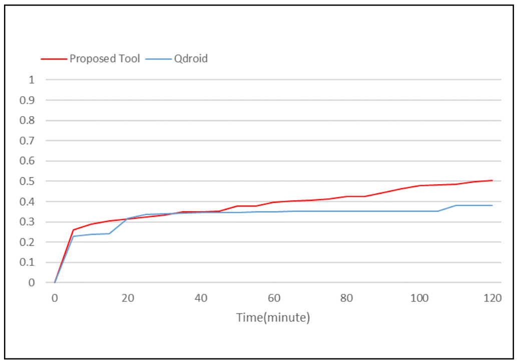

Figure 5. Coverage Evolution in Mileage

Second, we were only able to compare our tool with Qdroid. We need to compare our approach with other tools.

Third, in the experiments, we conducted three 2-hour trials for each app. However, there were some apps for which the coverage did not change at all after 2 hours. Since the execution time determines whether the speed of the search or the size of the search space is more important, it is possible that the results will change significantly by changing the execution time. In addition, there are apps where the coverage is almost the same each time, while there are also apps where the coverage fluctuated by more than 10%, suggesting that the number of trials may not be sufficient for some apps.

6. CONCLUSION

We applied reinforcement learning to test Android apps. Other researchers have done work on this, but we especially focused on (1) state definition so that the number of states does not explode, (2) a reward that changes depending on the situation, and (3) a learning approach to make full use of them. As a result, we were able to improve the coverage compared to Qdroid, which our implementation was based on.

Future work includes further evaluation and improvement of our proposed approach. As for the evaluation, more apps should be included in the experiment to make the evaluation more generalizable. We also expect that the performance of our approach can be improved by extending the supported UI actions and adjusting various parameters, states, and rewards.

ACKNOWLEDGEMENT

This work was supported by JSPS KAKENHI 22K11979.

References

[1] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? In *Proc. ASE 2015*, pages 429–440, 2015.

[2] Dong, Zhen and Bohme, Marcel and Cojocaru, Lucia and Roychoudhury, Abhik. Time-travel Testing of Android Apps. In *Proc. ICSE 2020*, pages 481–492, 2020.

[3] Hot Death. https://github.com/jpriebe/hotdeath Accessed 2023-1-17.

[4] D. Lai and J. Rubin. Goal-Driven Exploration for Android Applications. In *Proc. ASE 2019*, page 115–127, 2019.

[5] Logcat command-line tool. https://developer.android.com/studio/command-line/logcat Accessed 2023-1-17.

[6] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *Proc. ISSTA 2020*, pages 153–164, 2020.

[7] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. Deep Reinforcement Learning for Black-Box Testing of Android Apps. *ACM Trans. on Software Engineering and Methodology*, 31(4), Jul 2022.

[8] Tuyet Vuong, Shingo Takada. Semantic analysis for deep Q-network in android GUI testing. In *Proc. SEKE 2019*, pages 123–128, 2019.

[9] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey Accessed 2023-1-17.

[10] C. J. Watkins and P. Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[11] Write automated tests with UI Automator. https://developer.android.com/training/testing/ui-automator Accessed 2023-1-17.

[12] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static Window Transition Graphs for Android. In *Proc. ASE 2015*, pages 658–668, 2015.