2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC) | 978-1-6654-8303-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCWC54503.2022.9720807

Android GUI Test Generation with SARSA

Md Khorrom Khan
Computer Science & Engineering
University of North Texas
Denton, Texas 76203
Email: mdkhorromkhan@my.unt.edu

Renee Bryce
Computer Science & Engineering
University of North Texas
Denton, Texas 76203
Email: renee.bryce@unt.edu

***Abstract*—Android applications are often challenging to test because of large event spaces with an exponential number of event sequences. Several studies employ reinforcement learning to generate test suites in an effort to optimize code coverage and fault-finding effectiveness under limited testing budgets. In this paper, we generate test cases using the SARSA reinforcement learning algorithm for seven Android applications, each with a two-hour testing window. The SARSA generated test suites achieve 9.87% to 24.79% better line coverage, 6.9% to 20.09% better branch coverage, 7.88% to 28.48% better method coverage and 3.74% to 35.02% better class coverage than the test suites generated at random by the Monkey tool.**

***Index Terms*—Software Testing, Mobile Application Testing, Android Testing, Reinforcement Learning, SARSA**

I. Introduction

Mobile devices offer an easy means of communication and have become an integral element of our everyday life. The Android mobile operating system powers the majority of mobile devices, accounting for 83.8% of the Smartphone OS Market Share [1]. According to the "Global Developer Population and Demographic Study" by Evans Data, virtually half of the total mobile developers worldwide target Android as the first platform [2]. Android app developers publish an average of 1,957 mobile apps through the Google Play Store every day [3]. These applications are capable of performing complex tasks and processing sensitive information. Mobile applications demand a high level of credibility and dependability because the functions accomplished by an app rely on the quality of the application. Application quality is vital in the competitive Android market.

The Graphical User Interface (GUI) of an application is tightly coupled with the business logic to provide users with the functionality. Testing application functionality through the GUI is critical. Many previous research efforts focus on Android testing through the GUI [4], [5], [6], [7], [8]. Manual testing can be used to test the android application's GUI, but it is time-consuming. The Android platform supports mobile devices with different sizes of screen and versions of operating systems. Android applications are composed of activities, and activities can have multiple widgets. Testing all the available events and their combinations needs a substantial amount of time and manual effort. Automated testing may save time, cost and improve test coverage and efficiency.

Android UI/Application Exerciser Monkey [9] is a command-line utility, which promptly injects a pseudo-random sequence of events, based on screen coordinates, including touches, clicks, gestures, and some system-level events that comes with official Android application development framework. It is to note that, despite its popularity, Monkey lacks the ability to generate events by identifying application GUI elements; rather, it generates events by randomly clicking screen coordinates. As a result, Monkey often generates the same event multiple times or clicks on a non-interactive screen area. Sometimes it takes significant time for Monkey to reach functionalities of the application and, depending on testing resources, may result in subpar testing coverage.

In our previous work [10], we created an event selection strategy based on reinforcement learning algorithm Q-learning. It improves code coverage by exploring the application with trial-and-error interactions for intelligent event selection. For the identical set of test generation variables as input, an empirical evaluation of eight open-source native Android applications reveals that Q-learning outperforms random testing and achieves 3.31% to 18.83% more code coverage. In this paper, we make the following contributions to the study of reinforcement learning for Android Graphical Interface testing using the SARSA (State-Action-Reward-State-Action) algorithm.:

- adaptation of SARSA for Android GUI test generation and
- empirical evaluation of code coverage effectiveness of test cases generated by SARSA for seven Android applications

In the rest of this paper, Section II discusses the context and the current state of Android testing, Section III describes our test generation algorithm, Section IV explains the experimental design, Section V demonstrates the experiment results, Section VI discusses possible threats to validity and Section VII gives a conclusion.

II. Background and Related Works

The official integrated development environment for Android uses either Java or Kotlin as the programming language to develop native Android applications. Programming languages supported by cross-platform application development frameworks such as Iconic [11], React [12], and Flutter [13] include but not limited to Python, JavaScript and C++. Even though native Android applications are mostly written in Java, we cannot use traditional tools for testing standalone Java applications for Android. Android Applications differ significantly and manifest different types of bugs [14]. Android's distinct architecture combines and extends concepts from a variety of application domains, which include desktop, web-based, embedded and distributed systems [15]. The Android framework allows applications to support multiple input methods and gestures. These input methods and gestures need to be considered when developing test generation tools for Android applications. Dynamic UI management, system generated events, inter-app communications and context awareness are some of the other major challenges for Android test automation [15]. There has been ongoing research for years towards developing Android test automation tools [14], [16].

Model-Based Testing (MBT) tools [17], [18], [19] construct a high-level abstraction of the application under test (AUT) that describes the behavior of the AUT in terms of input, output, conditions, actions, and data flow from input to output. MBT tools use the application model created as finite state machines or state charts to generate test cases. Because of the complexity of Android applications, creating an accurate model takes time and can be error-prone. The test generation technique we present in this paper using SARSA does not need an AUT model to generate test cases.

Search-based techniques use meta-heuristic search optimization to generate test cases. EvoDroid [20] examines the source code of the AUT to create two different types of application models that describe the app's external interfaces and internal behaviors. EvoDroid applies a step-wise evolutionary algorithm with the goal of maximizing code coverage using these two models. SAPIENZ [21] detect faults and improve coverage by incorporating random-fuzzing with search-based exploration. Our application of SARSA generates test cases without requiring access to the application's source-code. We use source code only to instrument the application for collecting code coverage information. The test generation process is entirely black box and only requires the Android Package (APK) file.

Dynamic extraction techniques [22], [23] generate and execute event sequences on the fly at run-time. They do not need an abstract model of the AUT to generate test cases. Choi et al. [24] combined active learning with testing and proposed a machine learning approach called Swifthand to learn the AUT model dynamically. Their goal is to generate test cases to maximize branch coverage with minimal restarts. MacHiry et al. [23] presented Dynodroid, which uses a dynamic approach to generate a sequence of events using an "observe-select-execute" cycle. It observes the available events first, confirms their existence, and then chooses using a randomized algorithm. We adopted a dynamic event selection approach similar to Dynodroid in our previous work. Unlike Dynodroid, our previous work Autodroid [10] uses reinforcement learning-based algorithm Q-learning to select events. Q-learning applies a trial-and-error approach for intelligent event selection to improve code coverage. We use the same dynamic event selection approach to generate test cases using the SARSA reinforcement learning algorithm.

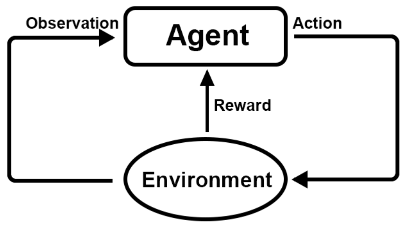

Fig. 1: Reinforcement Learning

The work in this paper uses Reinforcement Learning(RL) [25] to generate test cases without prior knowledge of the environment. Figure 1 shows a typical reinforcement learning setup. In a reinforcement learning framework, an autonomous agent takes an action in a particular state based on a behavior policy, observes the resulting state, and collects the immediate reward for taking the action. Agent receives a positive reward for taking measurably good actions and negative scores for measurably bad actions. From these observations, the agent updates the action value Q using a policy known as target policy or update policy that guides which action to take next in order to get the optimal cumulative reward. Popular reinforcement learning algorithms include but are not limited to the Monte Carlo method, Q-learning, SARSA, Q-learning - Lambda, SARSA - Lambda [25].

Several researchers use the reinforcement Q-learning algorithm for GUI testing. AutoBlackTest [26] is perhaps the first Q-learning based GUI testing solution for Java/Swing desktop applications. Esparcia-Alcázar et el. [27] propose TESTAR, a GUI testing tool based on Q-learning for desktop and web applications. Kim et el. [28] generate testcases with improved branch coverage for C applications using deep reinforcement learning with their tool Gunpowder. These tools are not applicable for mobile testing because of the unique characteristics of mobile applications, such as a wide range of screen sizes, OS versions, input methods and interaction mechanisms.

Our previous work proposes an Android GUI testing strategy using Q-learning to increase code coverage [10]. The optimized event selection using trial-and-error interactions employed in our technique accomplishes 3.31% to 18.83% better code coverage than random testing. Vuong et. al. [29] propose a similar test generation algorithm based on Q-learning for Android with a different reward function than our tool. In their approach, the discount factor was fixed at 0.9. Contrarily, our technique utilizes a variable discount factor based on the number of events in the subsequent state after executing an event, allowing the agent to look further ahead if a state has fewer events. QBE [30] explores GUIs and prioritizes the GUI action transitions using Q-learning to increase activity coverage and crash detection. Q-testing [31] is another Q-learning based tool that uses a curiosity-driven exploration strategy and a neural network to distinguish functional scenarios. Yasin et al. [32] present DroidbotX, which increases overall instruction coverage, method coverage, and activity coverage by employing Q-Learning with upper confidence bound (UCB) exploration. Andrea et el. [33] use a deep neural network in their tool ARES to achieve higher code coverage and detect faults in Android apps. DeepGUIT [34] applies Deep Q-Network for Android testing. This approach uses a neural network to approximate the action-value function using current information(states, actions, reward and following states).

The model-based tool AIMDroid [35] is the sole tool we found that uses the reinforcement learning algorithm SARSA for Android testing. Using a BFS algorithm, AIMDroid traverses activities and creates a BFS tree. Then it encases an explored activity in a 'cage' and employs a SARSA-guided fuzzing algorithm to explore the activity's inner-states. The primary goal of AIMDroid is to explore every activity and reduce activity transition time. Our work does not need to create any BFS tree. In this paper, we use SARSA to explore the UI space of the entire AUT instead of focusing on only one activity. Similar to our previous Q-learning algorithm, we systematically explore the unexplored areas of the app using SARSA with the goal of maximizing code coverage.

III. Proposed Approach

SARSA behaves and learns in accordance with the same policy and hence is known as an on-policy algorithm. In other words, SARSA uses the same policy to select an action and update the action value Q. The AUT serves as the environment. At each step, the SARSA agent selects a GUI event from the available events in a given state from the AUT. The agent executes the event, observes the reward, and updates the action value Q using the Q-value function [25] defined below:

$$Q(s,e) \leftarrow Q(s,e) + \alpha[R(e,s,s') + \gamma Q(s',e') - Q(s,e)] \tag{1}$$

where, $Q(s,e)$ on the left hand side is the new Q-value of event e after executing event and going to state s', $Q(s,e)$ on the right hand is the old Q-value of event e in state s, α is a hyperparameter called the *learning rate*, $R(e,s,s')$ is the immediate reward for taking event e in state s, $Q(s',e')$ is the Q-value of next selected event e' in the state s'. γ is known as the *discount factor*.

The value of learning rate α is typically set between 0 to 1. If the learning rate is 0, the Q value will never be updated and the agent will learn nothing. Learning will be quicker for a high value of α. We aim to learn the environment as quick as possible and set learning rate of 1 to maximize the learning. Hence, our Q-value function is:

$$Q(s,e) = R(e,s,s') + \gamma Q(s',e') \tag{2}$$

We adapt the definitions of state, event, reward function and discount factor from our previous work [10].

Definition III.1. A 3-tuple represents an action a: $a = (w,t,v)$, where w is a widget on a specific screen, t represents the type of action (e.g. click); v is arbitrary text for text field w. v is empty for all widgets that are not text fields.

Definition III.2. An n-tuple represents a GUI state s: $s = (a_1,a_2,a_3,...,a_n)$, where a_i denotes an action and n refers to the total number of unique actions on the screen.

Definition III.3. A set of one or more interrelated actions that can occur in a particular GUI state is referred to as an event. A 2-tuple $e = (s, A_e)$ is used to define it, with s representing a GUI state and A_e representing an ordered list of actions related to the event.

The reward function computes the immediate outcome of carrying out an event so that the test generation algorithm is able to distinguish between previously selected 'good' and 'bad' events. The immediate reward for executing event e in GUI state s is:

$$R(e,s,s') = \frac{1}{x_e} \tag{3}$$

where s' is the resulting state after executing event e and x_e is the execution frequency i.e. the number of instances event e has occurred. The events that have been explored previously are less likely to be explored again since the reward function is inversely proportional to the execution frequency.

The discount factor helps the agent look ahead and determine how the future rewards affect the Q-value function. For a state s' having a total number of events |E|, we define discount factor $\gamma(s',E)$ using equation 4.

$$\gamma(s',E) = 0.9 \times e^{-0.1\times(|E|-1)} \tag{4}$$

Typically, the value of the discount factor is within the range [0, 1]. $\gamma = 0$ makes the agent myopic and considers only the immediate reward. When the discount factor is 1, the agent will always give priority to future rewards. Instead of a static discount factor, we utilize a variable discount factor derived from the exponential decay function defined in equation 4. When there are few events in a state, our notion is that future rewards are more important and the agent should employ a large discount value.

Algorithm 1 illustrates the pseudocode for our test generation process using SARSA. As the test suite completion criterion, we use a two-hour time budget. Test case termination criteria is probabilistic to enable the agent to generate test cases of varying lengths. We use the ϵ-greedy exploration policy to select an event and update the Q-value. In this approach, the agent randomly selects an event with a probability of ϵ and the best event (the event with the maximum Q-value in a state) with a probability of 1-ϵ. The value of ϵ determines the classic problem of exploration vs. exploitation. The agent will select a random event most of the time if the ϵ value is high, and with a low ϵ value, the agent will exploit more, i.e., most of the time, the agent chooses a greedy event. In our approach, we hypothesize that the ϵ value of 0.3 is a good exploration vs. exploitation trade-off considering the large exploration space of Android applications. Algorithm 2 shows the pseudocode for ϵ-greedy event selection. It is the definition of *getEpsilonGreedyEvent* used in the Algorithm 1 line 11 and 29.

App Name	# LOC	# Branches	# Methods	# Classes
AnkiDroid	29063	11772	4091	500
Tricky Tripper	8244	2512	1766	290
Track Work Time	6403	2105	1211	174
The Kana Quiz	4453	2231	629	87
Tickmate	2654	770	395	60
SimpleReminder	1126	314	292	48
Open FNDDS Viewer	971	163	189	37

TABLE I: Characteristics of subject applications

IV. Experimental Setup

The experiments apply SARSA guided test generation to seven Android applications to examine the following questions:

Research Question 1: ***Is SARSA able to generate test cases with better code coverage than Monkey?***

Research Question 2: ***How does SARSA compare to Monkey in terms of code coverage progress over time?***

We use seven Android applications of various sizes and categories, downloaded from the open-source app repository F-droid [36] to evaluate our technique. AnkiDroid has the highest, and Open FNDDS Viewer has the lowest number of lines, branches, methods and classes. Table I shows the number of lines, branches, methods and classes for each of the subject applications. We instrument the applications to collect code coverage using free, open-source code coverage tool JaCoCO [37]. The test generation process installs the instrumented APK and runs the experiments. It uses Appium [38] and UIautomator [39] to extract the XML representation of the AUTs GUI and discover available widgets and actions identified by unique ID or XPath.

We utilize the random test generation tool Monkey that comes with Android Studio as a baseline to assess the performance of our algorithm. Monkey provides high code coverage and has been used as a baseline in many research studies. Typically, Monkey generates a single sequence for a test suite, while our tool generates multiple event sequences. We configured Monkey to run two hours and generate multiple event sequences for a fair comparison. We used the maximum event sequence length for each application generated by SARSA as the input for the Monkey

```
input  : $AUT$
input  : completion criterion, $c$
input  : initial Q-value, $V_{init}$
input  : test case termination criterion, $t$
output: test case, $TC$
output: test suite, $T$
1  begin
2      while not $c$ do
3          start AUT;
4          $TC \leftarrow \phi$;
5          $events \leftarrow$ getAvailableEvents();
6          foreach event in events do
7              if execFrequency($event$) = 0 then
8                  setEventValue($event$, $V_{init}$);
9              end
10         end
11         $eventTemp \leftarrow$
             getEpsilonGreedyEvent($events$);
12         while not $t$ do
13             $eventToExecute \leftarrow eventTemp$;
14             execute $eventToExecute$;
15             $TC \leftarrow TC \cup eventToExecute$;
16             if $eventToExecute$ exits AUT then
17                 updateReward($eventToExecute$, 0);
18                 setEventValue($eventToExecute$, 0);
19                 break;
20             end
21             $newEvents \leftarrow$ getAvailableEvents();
22             foreach event in newEvents do
23                 if execFrequency($event$) = 0 then
24                     setEventValue($event$, $V_{init}$);
25                 end
26             end
27             $\gamma \leftarrow$
                 calculateDiscountFactor($newEvents$);
28             $reward \leftarrow$ getReward($eventToExecute$);
29             $eventTemp \leftarrow$
                 getEpsilonGreedyEvent($newEvents$);
30             $eventValue \leftarrow$
                 getEventValue($eventTemp$);
31             $qValue \leftarrow reward + \gamma \times eventValue$;
32             setEventValue($eventToExecute$,
                 $qValue$);
33         end
34         $T \leftarrow T \cup TC$;
35     end
36 end
```

Algorithm 1: Test Suite Generation

```
Input: available events in a given state, $events$
Input: epsilon value, $e$
Output: selected event, $eventToExecute$
1 $p \leftarrow random(0,1)$;
2 if $p < e$ then
3     $eventToExecute \leftarrow selectRandom$(events);
4 else
5     $eventToExecute \leftarrow getMaxValueEvent$(events)
6 end
7 return $eventToExecute$
```

Algorithm 2: ϵ- greedy event selection

App Name	**Line**			**Branch**			**Method**			**Class**		
	SARSA	Monkey	Improvement	SARSA	Monkey	Improvement	SARSA	Monkey	Improvement	SARSA	Monkey	Improvement
AnkiDroid	39.62	14.83	**24.79**	25.16	8.37	**16.79**	49.36	20.88	**28.48**	67.3	32.28	**35.02**
Tricky Tripper	36.61	24.97	**11.64**	20.27	12.45	**7.82**	42.2	28.51	**13.69**	55.07	41.96	**13.11**
Track Work Time	58.13	36.52	**21.61**	38.69	22.56	**16.13**	63.29	39.58	**23.71**	81.09	52.01	**29.08**
The Kana Quiz	63.3	49.75	**13.55**	46.71	39.81	**6.9**	75.28	63.69	**11.59**	89.66	80.49	**9.17**
Tickmate	78.02	53.92	**24.1**	57.87	36.97	**20.09**	81.06	60.23	**20.83**	86.34	65.67	**20.67**
SimpleReminder	67.18	57.31	**9.87**	48.89	37.83	**11.06**	66.51	58.63	**7.88**	81.87	78.13	**3.74**
Open FNDDS Viewer	90.15	71.12	**19.03**	69.69	55.92	**13.77**	93.07	71.91	**21.16**	100	82.7	**17.3**

TABLE II: Average code coverage achieved by SARSA and Monkey testsuites

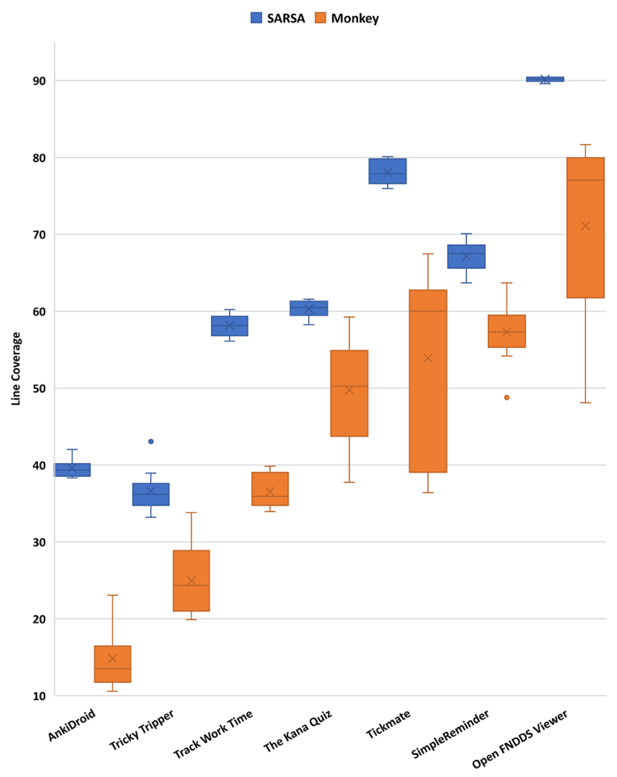

Fig. 2: Line Coverage across all the subject applications and for all runs

event sequence. Our Monkey script uses a unique seed as input for each event sequence to prevent sequence repetition. We run SARSA and Monkey experiments ten times on each application, with a two-hour time limit for each run.. Our test case termination criteria is probabilistic with a probability value of 0.05. Test case terminates by clicking on the Home button. We use two-second delays after each event to ensure that the next event is not executed before the AUT responds. We use two machines with identical configurations to generate the test cases, Ubuntu 20.04.3 LTS with 32 GB Ram and Pixel_3a emulator with API 29 (Android 10.0).

Parameters	**Monkey**	**SARSA**
Test Suite Generation Time (in hours)	2	2
Total number of test suites for an app	10	10
Delay between actions (in seconds)	2	2
Test case termination probability	0.05	0.05
Action Value Q (Initial)	-	500

TABLE III: Input Parameters for both SARSA and Monkey

We set a large starting Q-value of 500 for SARSA; this value will always be larger than the values the agent derives after interacting with the environment, and the agent will select every event in a given state at least one time. Table III shows our test generation parameters. After executing ten runs for each application, we collect code coverage and report the average.

V. Results and Discussion

Table II represents the average line, branch, method and class coverage achieved by SARSA and Monkey across ten runs for the seven subject applications. It also shows the code coverage improvement by SARSA over Monkey.

Research Question 1: *Is SARSA able to generate test cases with better code coverage than Monkey?* SARSA outperforms Monkey in all subject applications in terms of line, branch, method, and class coverage. Line coverage improvement ranges from 9.87% to 24.79%. AnkiDroid, which is the largest of our subject applications in terms of lines of code, achieved the highest line coverage improvement. It also achieves the highest method and class coverage improvement. The method and class coverage improvements range from 7.88% to 28.48%, and 3.74% to 35.02%, respectively. SimpleReminder, achieves the lowest line, method and class coverage improvement. It has 1126 lines of code, making it the second smallest of our subject applications. After a close inspection of the source code of SimpleReminder, we found that the uncovered lines of code are mostly related to some services that trigger based on context changes such as system broadcast of time. Our tool only works on GUI events, and it does not support context and system events yet. Monkey can generate several system-level events, despite that, our approach achieves higher coverage than Monkey.

The highest branch coverage improvement 20.09 is achieved by Tickmate, while The Kana Quiz achieves the lowest branch coverage improvement. Branch coverage improvement ranges from 6.9% to 20.09%. Table I shows that The Kana Quiz has a relatively high number of branches considering its total number of lines of code. It is a quiz application, and the quiz questions cover most of the code. The questions populate in the same activity. Since most of the possible events of this application are around the same activity and the app has high lines of code to branch ratio, Monkey can achieve a relatively high branch coverage. Therefore the branch coverage improvement by SARSA is lowest for this application. Open FNDDS Viewer

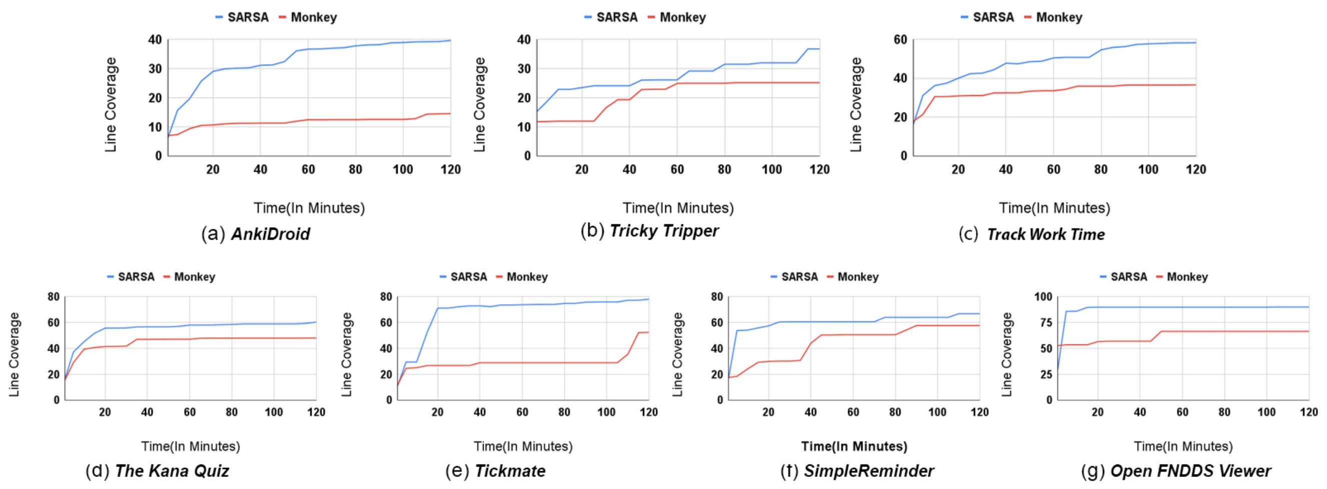

Fig. 3: Code coverage progress over time

is the smallest app we used with only 971 lines of code, and it achieved 100% class coverage by SARSA.

Figure 2 shows the distribution of line coverage achieved by both SARSA and Monkey in boxplots for all ten runs across all the subject applications. Branch, Method and Class coverage follow similar patterns. SARSA achieves higher minimum, median, and maximum coverage than Monkey consistently for all the subject applications. In all the applications, Monkey achieves the minimum coverage value. SARSA boxplots are comparatively shorter than Monkey for the same application in all the cases, implying that the code coverage achieved by SARSA for an application across ten runs has less variation than Monkey.

Research Question 2: ***How does SARSA compare to Monkey in terms of code coverage progress over time?*** To evaluate the performance of both SARSA and Monkey over time, we investigated line coverage progress every 5 minutes for each subject application. Due to space limitations, we graph the run among the ten runs that achieves the line coverage closest to the average value for each application and plot the line coverage graph in Figure 3. Other runs show similar patterns. The test generation time in minutes is plotted on the horizontal axis, and the vertical axis represents the achieved line coverage. We observe an early jump in small-sized applications, i.e., Open FNDDS Viewer and Simple Reminder. They reach maximum or almost close to maximum line coverage value by SARSA within 20 minutes. Similar patterns are observed in Tickmate and The KanaQuiz. AnkiDroid, Tricky Tripper and Track Work Time are the larger applications among our test subjects in terms of lines of code. With these applications, SARSA outperforms Monkey from the beginning, but there is no early jump, and the progress is more pronounced. After reaching a certain point, Monkey does not explore the AUT much but SARSA keeps exploring, given our strategy and parameters that encourage exploration. For all applications, SARSA-generated test suites achieve a faster rate of the line coverage.

VI. Threats to Validity

The choice of ϵ value could affect the study's validity. Selecting parameters for machine learning algorithms is always challenging. For the exploration vs. exploitation trade-off, we had to choose an ϵ value which could change the results. We used an ϵ value of 0.3 with an effort to balance exploration vs. exploitation. Future work will examine our technique with various ϵ values to understand trade-offs in this area better. We used seven Android applications to evaluate our technique. This might not be enough to generalize the results, since different applications have unique characteristics, and the results may differ. We sought to reduce this risk by selecting applications of varying sizes and categories. Our test case termination criteria are probabilistic, and the randomness of clicking the Home button can be a threat to validity. We ran ten trials for each application and collected the average results to mitigate this risk.

VII. Conclusions

Android applications are event driven systems that often have a large event space, and the nature of exponential event combinations poses challenges to testing budgets. In this paper, we demonstrate an automated test generation technique based on the SARSA reinforcement learning algorithm. We evaluate our technique compared to the popular test generation tool Monkey for seven native Android applications. SARSA outperforms Monkey for all the subject applications in line, branch, method, and class coverage for the same set of test parameters. Even though our technique does not support system events as Monkey does, SARSA achieved 9.87% to 24.79% better line coverage, 6.9% to 20.09% better branch coverage, 7.88% to 28.48% better method coverage, and 3.74% to 35.02% better class coverage than Monkey. SARSA consistently achieves higher minimum, median, and maximum coverage than Monkey for all runs across all the subject applications. Close inspection of the coverage progress over time shows that SARSA achieves higher code coverage than Monkey and does this at a faster rate of coverage. Future work will explore SARSA for Android test generation with different values of ϵ and discount factors, and compare with Q-learning across a broader set of applications.

References

[1] IDC: Smartphone Market Share - OS (2020) IDC. [Online]. Available: https://www.idc.com/promo/smartphone-market-share [Accessed: Dec. 30, 2021]

[2] Evans Data (2016). Global Developer Population and Demographic Study 2016. [Online] Available: https://evansdata.com/press/viewRelease.php?pressID=244 [Accessed: Dec. 10, 2021]

[3] 42 Matters: Google Play Statistics and Trends 2021. [Online] Available: https://42matters.com/google-play-statistics-and-trends [Accessed: DEC. 31, 2021]

[4] Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., & Memon, A. M. (2014). MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32(5), (pp. 53-59).

[5] Song, Wei and Qian, Xiangxing and Huang. (2017) EHBDroid: Beyond GUI Testing for Android Applications*In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering(ASE) (ASE 2017)* (pp. 27–37)

[6] T. Su. (2016) FSMdroid: Guided GUI Testing of Android Apps. *In Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), 2016*, (pp. 689-691)

[7] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. (2017). PATDroid: permission-aware GUI testing of Android. *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery. (pp. 220–232)

[8] Yuzhong Cao, Guoquan Wu, Wei Chen, and Jun Wei. (2018). CrawlDroid: Effective Model-based GUI Testing of Android Apps. *In Proceedings of the Tenth Asia-Pacific Symposium on Internetware (Internetware '18)*. Association for Computing Machinery, Article 19. (pp. 1–6)

[9] Google Inc., "UI/Application exerciser Monkey." [Online]. Available: http://developer.android.com/tools/help/monkey.html [Accessed: Dec. 31, 2021].

[10] 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 2018)*. Association for Computing Machinery, New York, NY, USA, (pp. 2–8).

[11] Iconic-Cross plaform mobile app development framework. [Online] Available: https://ionicframework.com/ [Accessed: Dec 31, 2021]

[12] React Native-A framework for building native apps using react. [Online] Available: https://reactnative.dev/ [Accessed: Dec 31, 2021]

[13] Flutter (2021). Beautiful native apps in record time. [Online] Available: https://flutter.dev/ [Accessed: Mar 8, 2021]

[14] Choudhary, S. R., Gorla, A., & Orso, A. (2015). Automated test input generation for android: Are we there yet? *In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 429-440).

[15] K. Rubinov and L. Baresi (2018). What Are We Missing When Testing Our Android Apps? *In Computer, vol. 51(4)*, (pp. 60-68)

[16] Samer Zein, Norsaremah Salleh, and John Grundy. (July 2016). A systematic mapping study of mobile application testing techniques. J. Syst. Softw. 117, C (pp. 334–356)

[17] Y. Baek and D. Bae (2016). Automated model-based Android GUI testing using multi-level GUI comparison criteria. *In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 238-249)

[18] Espada, Ana Rosario and Gallardo, María del Mar and Salmerón, Alberto and Merino, Pedro (2015). Using model checking to generate test cases for android applications. *In Proceedings MBT 2015. EPTCS 180*, (pp. 7-21)

[19] 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)*, Association for Computing Machinery (pp. 245–256)

[20] Riyadh Mahmood, Nariman Mirzaei, and Sam 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)*, Association for Computing Machinery, New York, NY, USA, (pp. 599–609)

[21] Ke Mao, Mark Harman, and Yue Jia. (2016). Sapienz: multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016), Association for Computing Machinery, New York, NY, USA, (pp. 94–105)

[22] Carino, S. (2016). Dynamically Testing Graphical User Interfaces. *In Electronic Thesis and Dissertation Repository, 3476. Western Libraries - Western University*

[23] 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)*, Association for Computing Machinery. (pp. 224–234)

[24] W. Choi, G. Necula, and K. 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 & Applications, OOPSLA '13*, (pp. 623–640)

[25] Sutton, Richard S and Barto, Andrew G (1988), Reinforcement learning: An introduction, *MIT press*

[26] L. Mariani, M. Pezzè, O. Riganelli and M. Santoro (2011). AutoBlackTest: a tool for automatic black-box testing. *In 2011 33rd International Conference on Software Engineering (ICSE)* (pp. 1013-1015)

[27] Esparcia-Alcázar A.I., Almenar, F., Martínez, M., Rueda, U., Vos, T. (2016). Q-learning strategies for action selection in the TESTAR automated testing tool. *In Proceedings of the 6th International Conferenrence on Metaheuristics and Nature Inspired Computing (META 2016)*, Marrakech, Morocco, 27–31 October 2016, (pp. 130–137)

[28] Kim, J., Kwon, M., & Yoo, S. (2018). Generating test input with deep reinforcement learning. *In 2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)* (pp. 51-58). IEEE.

[29] 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. ACM, (pp. 31-37)

[30] Y. Koroglu et al. (2018) QBE: QLearning-Based Exploration of Android Applications. *In Proceeding of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018*, (pp. 105-115)

[31] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li (2020). Reinforcement learning based curiosity-driven testing of android applications. *In Proceedings o f the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020*, (pp. 153-164)

[32] H. Yasin, S. Hamid, and R. Yusof (2021). Droidbotx: Test case generation tool for android applications using Q-learning. *Symmetry 2021, 13(2)*, p. 310.

[33] Romdhana, A., Merlo, A., Ceccato, M., & Tonella, P. (2021). Deep reinforcement learning for black-box testing of android apps. *arXiv preprint arXiv:2101.02636*

[34] Collins, E., Neto, A., Vincenzi, A., & Maldonado, J. (2021). Deep Reinforcement Learning based Android Application GUI Testing. *In Brazilian Symposium on Software Engineering*, (pp. 186-194)

[35] T. Gu et al. (2017). AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications. *In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (pp. 103-114)

[36] F-Droid. F-Droid - Free and open source Android app repository, F-Droid Limited, Available: https://f-droid.org/ [Accessed: Dec 31, 2021]

[37] JaCoCo - JaCoCo Java Code Coverage Library, Available: https://www.eclemma.org/jacoco/index.html [Accessed: Dec 10, 2021]

[38] Appium. Appium: automation For apps, Available: http://appium.io/ [Accessed: Dec 03, 2021]

[39] UIAutomator, Available: https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html [Accessed: Dec 31 2021]