Q-Learning based Reinforcement Learning for Android GUI Testing

Darian Jennings
Junior
St. Mary's University
San Antonio, Texas, 78228

Pisith Sork
Senior
Albany State University
Albany, Georgia, 31705

Tavis Jackson
Senior
Albany State University
Albany, Georgia, 31705

Corey Pieper
Senior
Macalester College
Saint Paul, Minnesota, 55105

Rony Raju
Senior
University of North Texas
Denton, Texas, 76207

I. Abstract

With the increasing sophistication of technology, the prevalence of Graphical User Interfaces (GUI) also increases. It serves as the connection between the user and application functions, so it stands to reason that it is the most significant basis for system level testing. Despite this, most forms of GUI testing are inherently flawed, as they require a great deal of manual operation and attention. Automation of GUI testing is still in its advent and often relies on random algorithms to perform actions that may result in crashes. This experiment proposes a more robust and fully automated structure that allows for a progressively more effective and accurate approach to testing and detecting faults in GUI for applications. The approach involves using a mobile-application crawler in tandem with a machine learning algorithm known as Q-learning in order to offer the best coverage-based testing that still uses random actions to pursue the best chances for finding faults in the GUI.

II. Introduction

As of June 2020, there are currently over 2.9 million Android applications available on the Google Play Store (1). Due to the wide success of the Android market and its open-source nature, increasing numbers of developers have moved to the Android platform. In the period of 2017 to 2018, out of the 105 billion apps downloaded, there were 75.7 billion downloads for Android and 29.6 billion downloads for iOS (2). The reason for the preference of Android development is due to developers often feeling "restricted" by the limitations of the tools and guidelines of the iOS market(3). The freedom that a lot of developers have on the Android platform allows them to express their creative ideas to their fullest content.

This is not without its costs however, since more freedom in development generally results in more vulnerabilities within the infrastructure of their creations. This is the main reason why many consider the applications that comprise the iOS platform to be more "stable" (3). Furthermore, many software engineers struggle to find cost-effective solutions to debug their applications before release (4). As a result, Android applications are more likely to be released to the public prone to bugs, exacerbating the general instability of the app market on a daily basis. One of the more prevalent reasons for this difficulty is that most developers are not familiar with the Android development platform and the Java technologies associated with them (4). To remedy their lack of experience and the time constraints for debugging, developers often rely on testing tools of dubious reliability. In an attempt to address the flaws in other algorithms and provide a better solution for testing, we constructed a Reinforcement Learning algorithm designed to help developers identify crashes and/or vulnerabilities much easier.

III. Background & Related Work

When it comes to identifying crashes and/or abnormalities within android mobile applications, there are a plethora of tools available to developers. Some use random test generation tools while others use automated test tools. Although these tools are widely available, many developers feel pressured to release an application to the market within a short amount of time, thus testing in general is often overlooked or minimal. This section of the paper highlights some of the most popular testing tools that developers have access to and why Q-Learning has the potential to be superior to these regarding time complexity and efficiency.

Monkey

Monkey is a stressing tool developed by Android that executes a sequence of randomized events (5). This process provides developers with an easier way of finding faults within the application. However, this tool also often limits the possible number of bugs that can be discovered. It can output high code coverage, but has great difficulty in finding obscure errors that are hard to replicate, since it is limited in detecting specific error types. This is where research comes into play, as many researchers have been exploring alternative methods to algorithms that can create higher code coverage, as well as detecting different, more specific types of errors.

Android Ripper

One algorithm of note in this regard is AndroidRipper, an automated testing technique that can be implemented as a tool for Android testing (4). This technique uses GUI ripping to automatically and systematically run through the application's UI to generate and execute test cases as new events arise (4). AndroidRipper employs Monkey in order to generate these test cases around the Activity component of the application. The key difference is that AndroidRipper does not execute these events at random, but instead decides whether an event is viable to be executed based on the exploration criterion. Allowing for higher code coverage and the encounter of states of events that would normally be inaccessible. AndroidRipper among other types of model-based techniques have distinct drawbacks such as having generally long runtime. As it incurs a higher cost to execute every combination of events possible within the combinatorial suite as more options are available in each particular GUI state. The other major flaw is that this technique has the tendency of creating often unachievable test cases that cannot be executed in the application under testing (AUT), thus this component of the tool becomes unreliable at times. With high run time and test cases that are flawed in execution, AndroidRipper is slowly becoming an unreliable testing tool for developers to implement.

Reinforcement Learning

Most forms of machine learning use a method called supervised learning, which uses a large data set that the algorithm analyzes in relation to their current environment. This method fundamentally uses the data set as a standard to train the algorithm to reach these desired conditions. Reinforcement Learning is a different type of machine learning algorithm that uses an agent to analyze an environment and conduct a random action that yields a set reward value based off of a fixed formula tailored to the specifications of the intended favorable results (8). This essentially allows the algorithm to train itself and use the formula as a standard instead of a data set. This would be a preferred method in cases where large data sets are unachievable or inefficient to produce. Reinforcement Learning also allows the algorithm to progress and perform its tasks better than what is humanly possible and achieve theoretically ideal results for performance.

Q-Learning

Q-Learning is a type of Reinforcement Learning technique where the agent learns from its environment through trial-and-error interactions (7). Similar to AndroidRipper it executes certain events in some logical manner and in Q-Learning based RL this is determined by a reward system awarding the crawler when a new event is to be executed; this allows the algorithm to explore the AUT in order to reach these more complex states that potentially have errors. Q-Learning is a solid solution when it comes to Android mobile application testing; in our research we use Q-Learning as our algorithm in order to detect and document errors and crashes as we crawl through Android application GUI states.

In regard to code block coverage, results have shown that Q-learning achieves about a 10.30 % higher average code block coverage compared to randomized testing across multiple open-source applications from the F-droid open-source repository (6). Q-Learning is a viable algorithm for mobile-application testing due to the RL aspect of the algorithm; with event Q-values and a reward system that dynamically adapts as the agent crawls through the application GUI states, Q-learning is not only an efficient method but also a reliable method that can produce useful data for developers as they test their applications for bugs and crash errors.

Appium

Appium is an open-source tool that we used for automating our mobile Android application testing. Since Appium is a cross-platform tool, this allows for our code to be implemented on various platforms such as Windows, iOS, and of course Android. The feature of Appium we were interested in was the WebDriver API, this allowed the bridge between our IDE (PyCharm) and our emulator.

Android Studio & PyCharm

Android Studio was used to create a virtual device that we could use for testing our RL crawler algorithm. Another aspect we were able to apply to our project was modifying the source code for JaCoCo instrumentation, to generate a code coverage file after our Q-learning crawler implementation finished. Furthermore we were able to run the code coverage data directly from Android Studio and the emulator was used to provide visual feedback of the crawler event sequences. PyCharm was used as our IDE to write the algorithm due to its simplistic nature and ease of access in bridging Appium and Android Studio all together via desired capabilities and the Appium-Python Client.

IV. Methodology

Through our Q-learning based RL crawler the agent is able to interact with its environment through a cumulative reward based event sequence execution. As the agent interacts with each event it identifies each event with a particular action type, unique stateID, as well as key Q-learning functions such as reward, Q-values, and times executed. With these unique event actions and stateID hashes, the agent is able to efficiently differentiate each event and execute appropriately. With this in mind the Q-learning based functions tie in together to enable the agent to discover and execute event sequences that lead to a higher reward.

After an application has been developed the testing phase is often tedious and consists of time consuming work. Our algorithm offers an efficient and consistent method for crash detection within application GUI states. With this algorithm a developer may test their application by simply altering the appPackage and appActivity within the algorithm code itself to satisfy their particular application names. A simple adb shell command was used to derive this information for each of the applications we tested. This technique is dedicated towards testing applications that have been fully developed and built rather than a tool to use while the app development is in progress.

A. Q-Value Function

The Q-value function we defined is the primary function used to implement the Q-learning behavior and allows the agent to "look ahead" at future events in the subsequent state. The function is based upon the immediate reward from executing the particular event that was selected and the expected future reward from the resulting state. With this the Q-value function also incorporates the discount factor γ multiplied by the maximum Q-value of the available events $e*$ in the new state $s*$. By explicitly basing this function upon which event the agent selects it allows the agent to gravitate towards events that have not yet been executed. The initial Q-value for events that have never been executed is set to a value of 500. This high initial value is greater than any Q-value the agent will calculate during its execution since the Q-value and reward function are both decreasing functions, thus encouraging the agent to execute every event in a state at least once. Since the agent prioritizes events that have been executed fewer times, this causes the agent to visit partially explored states rather than states that have been completely explored. The Q-value function is defined as:

$$Q(e,s) = R(e,s) + \gamma \times max_{e \in E} Q(e, s') \quad (1)$$

B. Reward Function

The reward function is used to calculate the immediate reward for executing an event e. The reward value is inversely proportional to the number of times the event has been executed, Xe. Defining the reward this way generates a higher reward for events that have been executed for the first time; thus, similar to the Q-value function, the reward function promotes unexplored events in each particular GUI state. The reward function we used is defined as follows:

$$R(e, s) = \frac{1}{Xe} \quad (2)$$

C. Discount Factor Function

The discount factor is used by the Q-value function to modify the value of the maximum Q-value in the new state. This coefficient mediates the amount of influence future rewards have on the selection of events in the reinforcement learning algorithm. A high discount factor places greater importance on future rewards and encourages the agent to look ahead, while a low discount factor places less importance on future rewards and focuses on short-term rewards. We defined the function that calculates the discount factor as an exponential decay function in terms of the number of available events |E| in the new state *s*'. Therefore, the agent will prioritize future rewards over immediate rewards in new states where there are a small number of possible events. The discount factor function is defined below.

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

D. SQLite Database

In order to support the persistence of data across separate executions of the crawler algorithm and fast look-up times for retrieving event information we used a SQLite database to store events. The database contains a single table of six columns: RowNumber (integer), EventValue (text), EventKey (text), TimesExecuted (integer), Qvalues (real), and Reward (real). The EventKey stores the hash of the state corresponding to the event and the EventValue stores the type of action. TimesExecuted keeps track of how many times the event has been executed and is initialized to 0 when the event is first inserted into the table. Qvalues is the Q-value associated with that event and Reward is the last reward calculated for that event, initialized to 500 and 5 respectively.

E. crashLogger

The crashLogger is initialized to an empty list at the beginning of the algorithm. Each iteration an event is executed and is appended to another list - testCase. As the agent crawls through the application, if the event that was executed leads to the application crashing then that testCase, which contains all the events that led up to the crash, is appended to the crashLogger. Having two seperate lists allows us to reset testCase to an empty list after each crash is appended to crashLogger, thus allowing us to log multiple crashes for each application under testing, AUT.

F. Pseudocode

Algorithm 1: App Crawler

```
Input: application under test, $app$
Input: home button probability, $home\_prob$
Input: inital Q-value, $q_{init}$
Input: completion criteria, $done$
Output: A crash log detailing the events leading up to each crash
1  begin
2      $knownStates \leftarrow \emptyset$
3      $crashLogger \leftarrow \emptyset$
4      $testCase \leftarrow \emptyset$
5      while not $done$ do
6          launch $app$
7          while $true$ do
8              if $random(0, 1) \leq home\_prob$ then
9                  $selectedEvent \leftarrow HOME$
10             else
11                 $currentState \leftarrow$ getCurrentState()
12                 $currentEvents \leftarrow$ getAvailableEvents()
13                 if $currentState \notin knownStates$ then
14                     for $event$ in $currentEvents$ do
15                         setQValue($event, q_{init}$)
16                     $knownStates$.append($currentState$)
17                 else
18                     for $event$ in $currentEvents$ do
19                         getRowNumber(event)
20                 $selectedEvent \leftarrow$ getMaxValueEvent()
21             execute $selectedEvent$
22             $testCase$.append($selectedEvent$)
23             $appState \leftarrow$ queryAppState($app$)
24             if $appState$ == $CRASH$ then
25                 $crashLogger \leftarrow crashLogger \cup testCase$
26                 $testCase \leftarrow \emptyset$
27                 setQValue($selectedEvent$, 0)
28                 break
29             else if $appState$ == $BACKGROUND$ then
30                 setQValue($selectedEvent$, 0)
31                 break
32             $newEvents \leftarrow$ getAvailableEvents()
33             $reward \leftarrow$ getReward($selectedEvent$)
34             $gamma \leftarrow$ calcDiscountFactor($newEvents$)
35             $maxQValue \leftarrow$ getMaxValue($newEvents$)
36             $qValue = reward + gamma \times maxQValue$
37             setQValue($selectedEvent, qValue$)
```

Lines 2-4 begin with creating empty lists for 3 specific variables: *knownStates*, *crashLogger*, and *testCase*. *knownStates* in line 2 will contain all of the states acquired throughout the run. *crashLogger* in line 3 will be used to retrieve a list of all of the events and write them to a text file for documentation. *testCase* in line 4 is used to save all of the events that led up to the crash. Afterwards, a condition is set to determine if the home button will occur. Lines 8 and 9 set the condition to if a random number from 0 to 1 is less than the *home_prob* (.05), then the selected event will become the home button. If the home button does not become the event, then the program retrieves the current state and all of the available events within that state in line 11-12. From there, lines 13-16 set the initial Q-Values for all of the events if the state is not known and appends the state to the *knownStates* list. If it isn't, then lines 18-19 retrieves the row number for later use. After that is complete, the program selects the event based on the which event in the state has the maximum Q-Value and executes that selected event. Then after appending the selected to the test case, the program retrieves the state of the application and uses it to determine if the app terminated or suspended. Lines 24-28 set the selected Event's Q-Value to 0 is the program terminated after selecting the event. This is done to help the program understand which events cause it to terminate and what would be the best selection. Lines 29-31 set the selected Event's *Reward* to 0. If the program is still running in the foreground after executing the event, the agent proceeds to receive all of the new available events and calculates the *Reward*, *Gamma*, and *Q-Value* functions in lines 32-36. Then proceeds to set the Q-Value for the selected Event and begins the new iteration of the loop.

V. Results

For this project, Traccar Client was used as the starting application due to its line count and lack of exit events that would cause the application to run into the background. After several tests, we found that Traccar Client was the most reliable out of all of the other apps. Table 1 shows a snippet of the results of Traccar Client and portrays how other tables would look within the database.

Table 1: Results of the Traccar Client Test

Traccar Client Results

Actions	Times Executed	Q-Values	Reward
[<actions.Click object at 0x035A0460>]	14	60.38	0.07
[<actions.Click object at 0x035A04D8>]	2	17.19	0.5
[<actions.Click object at 0x035A0400>]	1	91.85	1
[<actions.Click object at 0x035A0478>]	1	91.85	1
[<actions.Click object at 0x035A0418>]	2	17.19	0.5
[<actions.Back object at 0x035A04A8>]	1	0	0

As the program ran, it used the Q-Values to help with its selection for other events. Despite the high count for TimesExecuted for one event, the Q-Value still has the possibility of being higher than other events that have a low execution count. Furthermore, crashes are given a 0 to the Q-Value and Reward to notify the program to not select the event again as shown in the Table. Since the agent needs to learn which events it should and should not select, setting the main parameter that it uses to select a particular event to a value of 0 would cause it to not select the event action anymore and prioritize other events.

After a successful run was made for Traccar Client, a crashLogger was made to contain all the events that led up to the crash and testing shifted to other applications. The applications tested contain a variety of GUI states that the agent was able to interact with and explore. Derived from the F-Droid app repository, the applications used for testing ranged in line count from 2,609-265,235. *Greentooth* is an automatic Bluetooth disabler. *WhoHasMyStuff* is an inventory application that keeps track of items. *Trigger* is used for automating tasks on your phone. *Diagard* is a food library with a diabetes diary. *Enhanced Controller for Onkyo and Pioneer* is a sound profile management app. These applications were chosen due to the small amount of exit events. This allowed the agent to look for crashes rather than exiting frequently. The designated runtime for each application was 1.5 hours and the line count range was increased from 1000-5000 to above 5000 lines due to our positive results from Traccar Client. Table 2 shows all the results from testing of the other applications. The average number of events per crash case was calculated by taking the summation of the number of events executed for all the crash cases and dividing it by the total number of crash cases that particular application experienced during testing. We noted one application, Enhanced Controller for Onkyo & Pioneer, did not meet the designated runtime; this particular application was more complex in nature and featured GUI states that fluctuated upon user interaction. This agent was able to successfully interact with this application environment for about fifteen-minutes before this became an issue to the agent. Greentooth on the other hand was run for the entire time and the agent did not locate any crashes within the boundaries it explored. For our third application, WhoHasMyStuff, the agent was able to detect a plethora of crashes (111), it is likely that the application is still underdeveloped or not properly tested due to the agent finding such a high count of crashes within the time allocated. Based on our experience with the application Trigger, we noted that some of the functionalities within the application that led deeper into the GUI states seemed unstable upon user interaction, thus when running the algorithm the agent documented these as crashes, but was unable to explore past the initial GUI state the caused the particular crash.

Table 2: Crash Logger Results

Crash Detection (crashLogger)					
Applications	Line Count	Crashes Detected	States Accessed	Average Events Per Crash	Runtime (hours)
Greentooth	2,609	0	12	0	1.5
Traccar - Client	4,813	20	17	182/20 = 9	1.5
WhoHasMyStuff	6,962	111	18	225/111= 2	1.5
Trigger	9,357	7	12	79/7=11	1.5
Diagard	96,778	1	53	839/1 = 839	1.5
Enhanced Controller for Onkyo & Pioneer	265,135	4	16	130/4 = 32.5	0.25

After the results of the crashLogger output file were finalized, we noted 4 out of 6 applications generated a code coverage file. Greentooth, Traccar Client, and WhoHasMyStuff all managed to get a relatively high amount of code coverage with each line percentage being over 60%. However, Trigger's code coverage was very minimal, only average around 34% for the line code coverage. This shows that our algorithm requires more time for applications that have over 5000 lines of code and there needs to be more improvement for allowing the crawler to successfully traverse through the majority of the applications functions. Table 3 further exemplifies the results of the code coverage we collected.

Table 3: Code Coverage per Application

Code Coverage (JaCoCo)					
F-Droid Applications	Generated coverage file?	Line %	Method %	Class %	Branch %
Greentooth	Y	62	62	54	28
Traccar - Client	Y	100	75	78	50
WhoHasMyStuff	Y	94	67	63	37
Trigger	Y	34	31	21	14
Diagard	N	-	-	-	-
Enhanced Controller for Onkyo & Pioneer	N	-	-	-	-

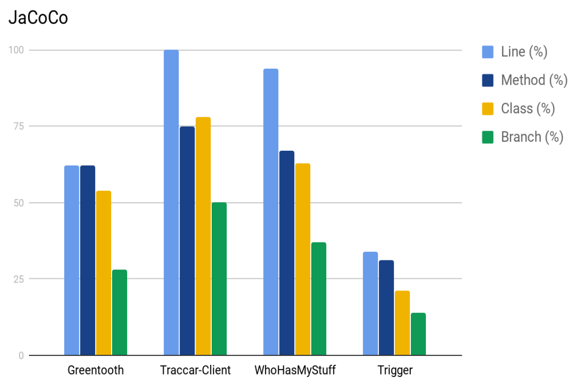

Figure 1: Graphical Representation of the Code Coverage

VI. Conclusion & Future Work

Exploring the GUI of an application can be an exhaustive task that may also lead to a decisional error on behalf of the user manually running these tests. Not only this but the |E|, total number of events in each particular state, may also be exhaustive depending on the application under testing, AUT. By adapting automated test generation we are able to simulate & automate the interactions a user would make via an emulator. The algorithm implements a crawling technique with an off policy Q-learning based Reinforcement Learning. This allowed the agent to select events based on particular reward and associated Q-values of the events in that particular state; optimizing the exploration of the application's GUI. In future work we will look towards higher algorithm efficiency, increased run time testing, and application in iOS emulators.

Many of these applications from the open-source app repository F-Droid have not been tested, thus each application is prone to bugs and errors that often lead to the application crashing. With this in mind there are a multitude of errors and AUT exits that must be accounted for such as the application accessing a webpage directly within the application still, events that open a different application, or even simple features that take the user to system settings. Writing a script to be all inclusive of every potential error across the board for all the applications is not only difficult in structure but rather time-consuming as well. It is easy to understand why developers skip this portion when releasing an application, the work is tedious and the cost of labor is high as well. Other limitations to the algorithm include applications with dynamic GUI states that fluctuate by user interaction. Future work will look to be inclusive of these dynamics and will look towards widening the range of mobile-applications successfully tested with this Q-learning based Reinforcement Learning crawler.

VII. References

1. Clement, J. (2020, June 17). Google Play Store: Number of apps. Retrieved June 25, 2020, from https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
2. Iqbal, M. (2020, June 23). App Download and Usage Statistics (2019). Retrieved June 25, 2020, from https://www.businessofapps.com/data/app-statistics/
3. iOS vs. Android: Which Platform is Best for App Development? (2019, July 19). Retrieved June 25, 2020, from https://liquid-state.com/ios-vs-android-which-platform-is-best/
4. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine and A. M. Memon, "Using GUI ripping for automated testing of Android applications," 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, 2012, pp. 258-261, doi: 10.1145/2351676.2351717.
5. “UI/Application Exerciser Monkey : Android Developers.” *Android Developers*, 27 Dec. 2019, developer.android.com/studio/test/monkey.
6. Adamo, D., Khan, M., Bryce, R., & Koppula, S. (2018, November 01). Reinforcement learning for Android GUI testing. Retrieved June 25, 2020, from https://dl.acm.org/doi/10.1145/3278186.3278187
7. Vuong, Thi Anh Tuyet, and Shingo Takada. “A Reinforcement Learning Based Approach to Automated Testing of Android Applications.” Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation - A-TEST 2018, 2018, doi:10.1145/3278186.3278191.
8. Sebastian Bauersfeld and Tanja Vos. “A reinforcement learning approach to automated GUIrobustness testing”. In 4th Symposium on Search Based-Software Engineering, page 7, 2012.