AUTOMATED GUI TESTS GENERATION FOR ANDROID APPS USING Q-LEARNING

Sreedevi Koppula

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2017

APPROVED:

Renee Bryce, Major Professor
Philip Sweany, Committee Member
Stephanie Ludi, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Victor Prybutok, Vice Provost of the Toulouse Graduate School
Mobile applications are growing in popularity and pose new problems in the area of software testing. In particular, mobile applications heavily depend upon user interactions and a dynamically changing environment of system events. In this thesis, we focus on user-driven events and use Q-learning, a reinforcement machine learning algorithm, to generate tests for Android applications under test (AUT). We implement a framework that automates the generation of GUI test cases by using our Q-learning approach and compare it to a uniform random (UR) implementation. A novel feature of our approach is that we generate user-driven event sequences through the GUI, without the source code or the model of the AUT. Hence, considerable amount of cost and time are saved by avoiding the need for model generation for generating the tests. Our results show that the systematic path exploration used by Q-learning results in higher average code coverage in comparison to the uniform random approach.
Copyright 2017

by

Sreedevi Koppula
I use this opportunity to express my sincere gratitude to Dr. Renee Bryce, for her excellent mentorship throughout the course of my thesis research. Her constant support and continuous encouragement helped me gain more insight into my research area. I am also very grateful for her consistent guidance and patience in helping me writing this thesis.

I would also like to thank David Adamo, a Ph.D. student, for his support and help throughout my thesis research. It is a great privilege to collaborate with him in this thesis, during which he always came up with and discussed interesting research ideas.
# TABLE OF CONTENTS

**LIST OF TABLES**

vi

**LIST OF FIGURES**

vii

**CHAPTER 1 INTRODUCTION**

1.1. Android Applications 1

1.2. Application Testing 3

1.3. GUI Testing of Android apps 4

1.4. Automated GUI Testing of Android Apps 4

1.5. Contribution of Thesis 5

1.6. Thesis Organization 6

**CHAPTER 2 RELATED WORK**

2.1. Capture-and-replay Tools 7

2.2. Model-Based-Testing (MBT) 8

2.3. Random Testing 9

2.4. Testing using Q-learning approach 9

**CHAPTER 3 ALGORITHMS**

3.1. Uniform Random (UR) 12

3.1.1. Algorithm 13

3.2. Q-learning 16

3.2.1. Reward Function $R$ 17

3.2.2. Action-Value Function $Q$ 17

3.2.3. Action Selection 18

3.2.4. Algorithm 19

3.2.5. Example 21

**CHAPTER 4 EXPERIMENTAL SETUP**

24
4.1. Applications under Test
4.2. Test Generation Parameters
4.3. Test Suites Generation

CHAPTER 5 RESULTS AND DISCUSSION
5.1. Code coverage
5.1.1. Block coverage
5.1.1.1. Block coverages of Budget app
5.1.1.2. Block coverages of Droidshows app
5.1.1.3. Block coverages of Moneybalance app
5.1.1.4. Block coverages of Tomdroid app
5.1.1.5. Block coverages across all the AUTs
5.1.2. Method coverage
5.1.3. Class coverage
5.2. Progressive Block Coverages
5.3. Test suites generation time
5.4. Implications for Software Testers

CHAPTER 6 CONCLUSION AND FUTURE WORK

APPENDIX A ARCHITECTURE OF TESTS GENERATION FRAMEWORK

BIBLIOGRAPHY
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Reward $r_{init}$ and Action value $actionValue$ captured for each action in each episode</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Details of Android apps used in the experiment</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Tests generation parameters</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Average block coverage achieved using UR and Q-learning algorithms</td>
<td>27</td>
</tr>
<tr>
<td>5.2</td>
<td>Average method coverage achieved using UR and Q-learning algorithms</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Average class coverage achieved using UR and Q-learning algorithms</td>
<td>32</td>
</tr>
<tr>
<td>5.4</td>
<td>Average tests generation time (in hours)</td>
<td>35</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Three activities of Droidshows app</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Main activity of Droidshows app</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>Clicking HOME button exits the AUT</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Clicking BACK button exits the AUT</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Clicking EXIT button exits the AUT</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>Clicking Hyperlink moves the AUT to backend and opens Browser app</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Algorithm for Test Suite Generation using UR approach</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>Algorithm for Test Suite Generation using Q-learning approach</td>
<td>20</td>
</tr>
<tr>
<td>3.7</td>
<td>Android application A represented in terms of states and action</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Block coverages of the Budget app</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Block coverages of the Droidshows app</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Block coverages of the Moneybalance app</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Block coverages of the Tomdroid app</td>
<td>30</td>
</tr>
<tr>
<td>5.5</td>
<td>Block coverages across all AUTs</td>
<td>31</td>
</tr>
<tr>
<td>5.6</td>
<td>Progressive block coverages of the app Budget</td>
<td>33</td>
</tr>
<tr>
<td>5.7</td>
<td>Progressive block coverages of the app Droidshows</td>
<td>33</td>
</tr>
<tr>
<td>5.8</td>
<td>Progressive block coverages of the app Moneybalance</td>
<td>34</td>
</tr>
<tr>
<td>5.9</td>
<td>Progressive block coverages of the app Tomdroid</td>
<td>35</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Android has become the most widely used mobile platform in the world, with a market share of 86.8\%[5] and the number of Android apps is also increasing dramatically[4]. These apps possess large execution space and pose new challenges in the area of Software Testing. The testing techniques for Android apps are not matured yet. New technologies introduced many problems in Android app testing, which requires tools or other testing techniques. One specific area of problems is Effective Tests Generation. In our research, the main goal is to improve the quality of the tests generated for a given Android app. To achieve this, we propose and implement a reinforcement learning algorithm that is discussed further in this paper.

1.1. Android Applications

An Android app is comprised of screens called activities and the main activity is displayed to the users once the app is started. Figure 1.1 shows three activities of the Droidshows app and Figure 1.2 shows the main activity. Each activity has a varying number of widgets, which are objects on the User Interface (UI) of the app. The users perform actions such as click a button, type text in a text field, drag and drop an object, highlight a piece of text, etc. to interact with the app to accomplish their tasks. The widgets present in the activities of Figure 1.1 are rounded in red.

We used Appium [1] and UIAutomator [7] in our work, to retrieve the XML representation of an activity in the given AUT. Using this XML representation, we find widgets that are available in an activity and the types of user interactions that are enabled on these widgets. The widgets are uniquely identified by ID or XPath. A few terms involved in this work and their definitions are detailed below.

GUI Action: A GUI action \( A \) is denoted by a 3-tuple \( A = (w, t, v) \), where \( w \) is a widget present in an activity, \( t \) is the type of operation that is performed on the widget and \( v \) is the value of text entry. Widgets are classified into actionable and non-actionable types.
Figure 1.1. Three activities of Droidshows app

Figure 1.2. Main activity of Droidshows app

- Actionable widgets: The widgets on which some operation can be performed. Examples include buttons, hyperlinks, text fields, etc.
- Non-actionable widgets: The widgets on which the users cannot perform any operation. Examples of non-actionable widgets are text, labels, etc.

Types of operations that can be performed on a widget are:
- Click, long click, type text, etc. on actionable widgets.
- Read text, select text, etc. on non-actionable widgets.

The tuple \( v \) contains arbitrary text value if the widget \( w \) is a text field. For all non-text field widgets, the value of text entry is empty.

**GUI State:** A GUI state \( S \) is denoted by an n-tuple \( S = (A_1, A_2, A_3, \ldots, A_n) \), where \( A_i \) specifies the \( i^{th} \) GUI action that is available on a particular activity of the app and \( n \) specifies the total number of actions that are associated with the widgets present in an activity.

An Android app becomes more popular if it is functional, free from errors, reliable and contains an easy-to-use UI. Any app deviating from these features might not attract the users and might frustrate them by repeated crashes, application hang and other software bugs, which eventually might lead to the failure of the app, incurring a substantial loss to the application development firms and its users. Hence, any app must be tested before adding to the Play store.

1.2. Application Testing

Testing is performed to find bugs that are associated with the Application Under Test (AUT). Lack of testing may result in a huge loss to the human life and economy. Ariane 5 flight 501 failed due to the software failure, which caused a loss of $360 million [25] and Therac-25, a computer-controlled radiation therapy machine, overdosed six people due to software errors [26]. Hence, application testing is necessary to identify any areas of weaknesses in the system, to reduce future costs that might occur due to improper functionality of the system and to increase confidence in the system.

White-box and black-box testing are two different types of testing approaches that are commonly used to test any application. White-box approach tests the internals of the AUT, requires specific data inputs to test specific paths of the code and finds which area of code is functioning incorrectly [24]. In the case of black-box approach, the AUT is treated as a black box and is tested without any knowledge of its implementation [24]. This approach is very helpful for the testers who want to test third-party applications for which the implementation details are not known or whose source code is not available.
In this thesis, we use black-box testing approach. We do not need either the implementation details or the source code to test an AUT by using the testing techniques that are implemented in this paper. For our research, we need only the .apk files of the targeted apps.

1.3. GUI Testing of Android apps

For any Android app, GUI is a primary component, as it allows the users to access the entire functionality of the app by interacting with the widgets that are available on the app’s UI. GUI testing is used to find any security vulnerabilities, crashes and exceptions that occur while using the app. This requires the simulation of user actions on the app and is different from the traditional testing, where we test any app by invoking it’s code methods directly.

The GUI testing can be done using either of the two following testing methods: manual and automated. In manual GUI testing, no tools or scripts are used and the testing requires high human involvement. This is less accurate, unreliable and not practical when the test cases must be repeatedly executed. In the case of automated testing, the test cases are either written or generated and executed using tools or scripts. Hence, the automated testing is reliable, accurate and primarily helpful for repeated execution of application tests.

Several test automation tools such as Robotium, Appium, Ranorex, etc. are available to write and execute test cases for mobile applications, making the testing activity easier [17]. However, writing the test cases using tools and scripts for all the available and upcoming apps is impractical and inefficient in terms of time, effort and cost. Hence, we propose automated GUI testing approaches, which generate the test cases instead of requiring the users to write test cases for Android app testing.

1.4. Automated GUI Testing of Android apps

GUI test automation is a difficult task, as it needs to mimic the human interactions with the GUI by understanding the various widgets in the UI and their purposes. Two approaches to automated GUI testing are: static and dynamic. The static approach to testing is also called model-based-testing (MBT) [10], where, the GUI model of the AUT must be captured and stored in a specific format (Example formats are tree, finite state machines and XML). The extracted
models are used to generate a sequence of events by means of algorithms such as Depth First
Search (DFS), Breadth First Search (BFS), choosing a sequence of length four, etc. and store the
sequence of events in a file, where each file corresponds to one test case. The generated test cases
are run against the AUT in order to test it. However, the models constructed contain only abstract
details of the app’s GUI behavior and might not reflect the actual behavior [11]. As a result, the
test cases that are generated by using the model might be only partially executable.

In contrast, in the case of dynamic approach, the structure of the AUT is not required to
generate the test cases. This approach follows the strategy of observe-select-execute, where all
the GUI actions of the current state of AUT are observed; one action is selected based on the
selection strategy under consideration and the selected action is executed on the AUT. This process
of observe-select-execute strategy is repeated to generate a test case that contains a sequence of
dynamically chosen actions. In this approach, the extraction of the GUI model is not required and
any GUI changes in the app will not affect the testing, as this approach does not depend on the
GUI model of the app. Hence, in our research work, we adopt the idea of dynamic automated GUI
tests generation for testing Android apps.

1.5. Contribution of Thesis

Many tools and techniques are available for the GUI testing of Android apps. We discuss
few of the existing tools and techniques in Chapter 2. In this thesis, we mainly focus on generating
effective GUI test cases for any given Android AUT, in a dynamic manner. For this, we developed
a framework that dynamically generates GUI test cases based on a reinforcement learning algo-

rithm. We evaluate and compare the performance of this reinforcement learning algorithm against
a basic random approach to generate GUI test cases. The performance is measured in terms of the
code coverages that are achieved by each approach. Code coverage [21] is a measure to indicate
the amount of source code of an app that is executed when the app is tested. We also provide
the progressive coverage details, where the progressive code coverage specifies initially the code
coverage of the first 25 test cases, next, the coverage of the first 50 test cases, next, the coverage of
the first 75 test cases, and so on, until all test cases of a test suite are used in calculating the code
coverage. A high code coverage generally indicates a high chance of detecting the bugs that are
present in the app.

1.6. Thesis Organization

Following is the organization of this thesis: the literature works related to our research are presented in Chapter 2; the algorithms that are implemented in our framework are presented in Chapter 3; the experimental setup that is used to generate the tests and measure the efficiency of each approach is discussed in Chapter 4; the results obtained are presented in Chapter 5; the conclusion of this research and the possible future works are discussed in Chapter 6; and the architecture of our framework is explained in Appendix A.
Automated GUI tests are a quicker and cheaper means to test an app. A good amount of research is going on to facilitate the automated GUI testing of Android apps. In this chapter, we discuss a variety of literature works in the area of GUI testing for Android apps.

2.1. Capture-and-replay Tools

Capture-and-replay is one of the approaches to automated GUI testing. The capture module captures all the user interactions with the GUI in the form a file and the replay module uses the recorded file to replay the events.

Gomez et al. [15] presents RERAN, a record-and-replay tool for Android platform. It’s capture module captures the low-level GUI and sensor event stream, and the replay module replays the events accurately. It doesn’t require the source code of the app. Hu et al. [19] presented VALERA, a Versatile-yet-lightweight Record-and-replay tool for Android. It uses sensor-oriented replay technique to gain high accuracy in terms of finding the existing and new bugs of the AUT. Halpern et al. [18] presents Mosaic, a cross-platform record and replay tool for Android. User interactions with the AUT are captured in a human-readable format, which enables the users to edit the recording as per their requirements when necessary. These interactions captured on smartphones, emulators and so on, can be replayed on other devices. Kassila et al. [22] presents Testdroid, an online platform for GUI testing of Android devices. It allows the users to test their recorded test scripts on a variety of Android devices in parallel and to report the results to the users. Liu et al. [27] proposed a capture-and-replay approach, where the user events on Android apps are captured in the form of Robotium [35] test scripts that are later executed to replay the recorded events. It also supports the insertion of assertion statements into the test scripts when the capture module is capturing the user events.

The capture-and-replay tools capture the events that are triggered on the device and will replay the exact input events that were captured during the original recording. This technique may test the apps efficiently when no changes are made to the app’s UI or functionality. However,
when some randomness is introduced in the app or when slight UI changes are made, the original recording becomes invalid. Moreover, this approach requires considerable effort to produce a sufficient number and combinations of interactions for the users to manually interact with the AUT in order to record the effective test cases. In our work, we do not require any manual intervention with the AUT to generate test cases and any changes to the app’s UI does not affect our effort in generating the tests dynamically.

2.2. Model-based-testing (MBT)

Model-based-testing technique constructs and uses the model of the AUT as a base for generating the test cases. Manually, the GUI models are constructed by going through the specifications of the AUT or by interacting with the app directly, but, the manual construction of a model is error prone and is a time consuming activity. Hence, the automated extraction of GUI models for the subject AUTs is preferred.

Tools such as ORBIT [34] and MobiGUITAR [9] are available for automated extraction of GUI models for the Android apps. Amalfitano et al. [8] presents AndroidRipper, a tool that finds the model of the AUT by using ripping, a reverse engineering technique, where the AUT’s GUI is traversed to build the model of the app. The model constructed is then used to generate and execute the test cases offline. Takala et al. [32] discusses the solution for Android test automation by the usage of model-based testing toolset. The toolset includes tools for test modeling, design, test generation and test debugging. Jensen et al. [20] proposes an approach that uses the model of the AUT to generate a sequence of events for each target, where the target can be lines or branches in the source code of the app. This approach initially generates an event handler summary of the AUT and then uses the summary along with the model to generate the sequences of events. Griebe et al. [16] proposes a model-based approach to automate the testing of context-aware mobile apps. This approach uses Unified Modelling Language (UML) model that includes the control flows and context information of the AUT for test cases generation and execution.

Model-based-testing helps in generating test cases with a better code coverage and fault detection. However, the quality of the generated test cases depends on the model built and building a model requires significant amount of time. Any changes to the UI of the AUT requires rebuild of
the model, which is not a good sign when an app’s UI undergoes frequent changes. In our research, the effort and time taken to build a model is saved because our proposed approach does not require any model for tests generation.

2.3. Random Testing

Random testing is an approach where the Android AUT is initially launched on either an Android emulator or device; the set of all GUI actions that are available on the current state are identified; one GUI action is selected at random from the set and send to the AUT for execution.

UI/Application Excerciser Monkey [6] is a fuzz testing command-line tool that comes along with the Android Software Development Kit (SDK) [14]. This tool generates a stream of user actions such as type text, click, swipe, etc at random and sends them to the AUT running on Android emulator. However, Monkey does not convert the sequence of actions into test cases and the actions generated are very random. Machiry et al. [28] developed an automatic input generation system for Android apps called Dynodroid, which uses a randomized algorithm to select and execute a predefined number of events in a single sequence. It also implemented the following two systematic approaches: BiasedRandom and Frequency that differ in the way of choosing an action from the set of available actions. However, these approaches do not select actions in an intelligent way.

The random testing approaches do not explore the AUT systematically. As the actions are chosen at random, there is a high chance that the already selected actions are repeatedly selected that might eventually result in a lower code coverage. In our work, the proposed approach systematically explores and learns the AUT. It does an intelligent way of action selection, aiming to achieve a higher code coverage.

2.4. Testing using Q-learning approach

For any AUT, few actions are easier to access and few actions are not easily accessible as they are deeply nested. By using random algorithms, it is unlikely that the deeply nested actions are selected and executed. Moreover, the random algorithm might select the actions redundantly due to it’s random nature. To achieve the access to deeply nested actions and to select the pre-
viously lesser executed or unexecuted actions, we need to change the probability distribution of actions over the sequence space, which can be achieved by using the Q-learning \cite{33} approach. Q-learning is a reinforcement learning \cite{23} algorithm. An agent in a Q-learning environment learns an optimal action-selection policy through trial-and-error interactions. Based on the current state of the environment, the agent chooses an action and evaluates the resulting reward. In this approach, instead of randomly selecting the actions, the main goal of the agent is to learn a sequence of actions that maximizes the cumulative reward. In the context of Android app testing, AUT is the environment; each activity of the AUT is a state in the environment; GUI actions of an activity are the set of actions available in the current state of the environment; and the testing framework is the agent. Initially, the testing framework has no knowledge of the AUT. As the framework generates and executes test cases, the knowledge about the AUT is updated. The framework uses the knowledge gained to take efficient future action selection decisions.

Bauersfeld and Vos proposed Q-learning for automated GUI testing in 2012 \cite{12}. The main goal was to only discuss the Q-learning technique that changes the probability distribution over the sequence space. This paper does not include any implementation details of the Q-learning technique in testing Android apps and also does not provide any quantitative analysis to support the efficiency of Q-learning in automated GUI testing. Later in 2014, Bauersfeld and Vos proposed an advanced action specification and selection mechanism that uses Prolog specification in addition to Q-learning, to derive sensible and sophisticated actions that explore large and complex GUIs \cite{13}. Their implementation was limited to run on MacOSX only. Mariani et al. \cite{29} introduced AutoBlackTest, a tool that automatically generates test cases for the given AUT by using the Q-learning algorithm. The implementation of AutoBlackTest is limited to testing of Java/Swing applications and could be extended to any GUI framework supported by IBM Functional Tester, including JAVA, .NET and other Windows and Linux GUI frameworks, but it does not provide the implementation of Q-learning to generates test cases for the Android apps.

To the best of our knowledge, this work is the first implementation of Q-learning to automated GUI tests generation for Android apps. To improve the quality of automatically generated Android app test cases, this paper attempts to achieve intelligent action selection by using
Q-learning and action selection heuristics based on the domain knowledge of Android app. Specifically, this paper makes the following contributions:

- An adaptation of Q-learning approach to dynamic and automated GUI testing of Android apps.
- Evaluation of the proposed approaches in terms of the code coverage.
CHAPTER 3

ALGORITHMS

In this chapter, we discuss the following two tests generation algorithms that are implemented in our framework: a) Uniform Random (UR) and b) Q-learning. In each of these algorithms the basic procedure is comprised of the following three steps that repeat in order to generate the tests:

- Obtain the set of actions that are associated with the current state of the app.
- Select one of those actions based on the tests generation approach.
- Execute the selected action.

The main difference between the UR and Q-learning approaches lies in the strategy behind the selection of a GUI action from the set of available actions. The following sections explain these algorithms in detail.

3.1. Uniform Random (UR)

UR is a basic approach to automate the tests generation for Android apps. This approach is used as a baseline for evaluating the performance of Q-learning approach. All widgets existing in the current state of the AUT are identified and corresponding GUI action objects are created for each widget. A GUI action is selected at random from the set of available actions and executed on the AUT. Due to random selection, each action has an equal probability of being selected. Hence, some actions that require much attention may not actually obtain the necessary attention and are explored less, resulting in a lower code coverage and a lesser chance of finding bugs in the AUT. Also, using UR approach, it might take much time to explore all the activities of the AUT. To overcome these disadvantages that are associated with the UR approach, we implement the Q-learning, an approach that has the intelligence of selecting actions that are previously either lesser explored or unexplored.
3.1.1. Algorithm

The algorithm for UR based tests generation is given in Figure 3.5. The algorithm takes the configuration.json file and the .apk file of the AUT as inputs. The configuration file contains the details such as the absolute path of the .apk file of the AUT, the number of test cases $n$ that need to be generated by the framework, etc. In the algorithm, we mark the end of a test case when the AUT exits or moves to the background and we consider the AUT to exit or move to the background only under the following conditions:

- Clicking Android device’s HOME button will move the AUT to the background, as given in the Figure 3.1.
- Clicking BACK button, whose execution might display the HOME screen of the device, moving the AUT to the background, as shown in the Figure 3.2. The BACK button execution may not always exit the app. The app exit depends on the app’s state when the BACK button is clicked.
- Selecting and executing Exit/Close button action exits the application as shown in the Figure 3.3.
- Clicking a widget that is present on current app’s activity that opens another mobile app (e.g. Browser, Camera, Gallery) as given in the Figure 3.4.

The selection of HOME or BACK button is decided by the randomly generated probability value for the variables home_btn_prob or back_btn_prob respectively. The method GenerateRandomValue will generate a random value in the range $[0, 1]$. If the random value of home_btn_prob is less than or equal to 0.05 in line 8, it indicates that the algorithm needs to select HOME button on the Android emulator. Hence, the value of selectedAction is set to HOME in line 9. If the random value of back_btn_prob is less than or equal to 0.05 in line 10, the algorithm will need to select BACK button on the Android emulator. Hence, the value of selectedAction is set to BACK in line 11. If the random values of both home_btn_prob and back_btn_prob are greater than 0.05, all the available actions $A$ in the current state of the AUT are captured in line 13 by using the method getAvailableActions and one action from $A$ is chosen at random by using the method getRandomAction in line 14. The chosen action is assigned to the variable selectedAction. The getRandomAction
Figure 3.1. Clicking HOME button exits the AUT

Figure 3.2. Clicking BACK button exits the AUT

The generated pseudo-random number is used as an index for action selection from the list $A$. Upon the execution of $\text{selectedAction}$ in line 15, the $\text{selectedAction}$ details are appended to the current $\text{testCase}$ in line 16. If the AUT exits or if another application such as mobile browser, gallery app, etc. is opened as a result of line 15, then the algorithm identifies the scenario as the end of current
test case and adds the test case to the test suite \( T \) in line 21. Lines 7 to 20 are repeated until the AUT exits or moves to the background. Once a test case is generated, the counter of the test cases \textit{testCaseCount} is increased by 1. The lines 4 to 23 are repeated until the user specified number of test cases are generated by the framework.
3.2. Q-learning

Q-learning algorithm is discussed briefly in the section 2.4. In this section, we go into the formal details and look into the various terms that are associated with the algorithm. An Android app can be modeled as a finite Markov Decision Process (MDP), which is a discrete time stochastic control process [30]. It contains an agent, a finite set of states \( S \), a finite set of actions \( A \) in each state, a reward function and an action-value function. The agent learns an action-value function \( Q \) for all the available actions at a given state. The value of \( Q \) defines the likelihood of choosing a particular action. Q-learning is used to find an optimal action-selection policy for the given Android AUT, where the policy determines the rule that an agent must follow in action selection from a set of actions [33]. Action execution is followed by each action selection, which moves the agent from current state to a new state. The agent is provided with a reward \( r \) upon executing the action \( a \). The value of reward is determined using the reward function \( R \). The ultimate goal is to
maximize the total return reward of the agent over each successive step, starting from the initial activity of the AUT, which resembles the goal of exploring all possible activities and actions of the AUT by minimizing the selection of already chosen activities and actions.

The action-value function $Q$ and the reward function $R$ associated with the action selection process of Q-learning approach are discussed below.

3.2.1. Reward function $R$

Reward function $R$ is used to calculate the reward value that must be associated with the action $a$, whose execution transmits the state of the AUT from the current activity $s$ to a new activity $s'$. We define reward function in such a way that our tool differentiates between good and bad actions. Good actions are those that get a higher reward value than that of the bad actions. In our work, the reward function $R(a, s, s')$ is taken as follows:

$$R(a, s, s') = \begin{cases} 
    r_{init}, & \text{if } x_a = 0 \\
    \frac{1}{x_a} \times a_{s'}, & \text{otherwise}
\end{cases}$$

(1)

where $r_{init}$ is the initial default reward that is associated with each GUI action that has not yet been selected during the test generation process, $x_a$ is the number of times the action $a$ of state $s$ has been executed, and $a_{s'}$ is the number of actions in the state $s'$ that were not in the state $s$. Hence, the more often an action has been executed and results in a new state which contains fewer number of actions, the less desirable it will be for the agent.

By means of this reward function, our framework is likely to explore the actions that are previously less explored as they possess high reward values. Hence, it ensures that a particular action is not selected multiple times while exploring the AUT.

3.2.2. Action-value function $Q$

Action-value function $Q$ is used to calculate the action-value of an action $a$, which is present in a particular state $s$ of the AUT. It uses the value of current reward that is obtained for executing action $a$ and the optimal future value that is associated with action $a$. This function is essential as it enables our testing tool to look one step ahead when making a decision of what action is to be
selected in a particular state to favor the exploration of an app. The action-value function is defined as follows:

\[
Q(s, a) = R(s, a, s') + \gamma \cdot \max_{a^* \in A_{s'}} Q(s', a^*)
\]  

where \(Q(s, a)\) is the value of action \(a\) that is present in state \(s\), \(R(s, a, s')\) is the reward value for executing action \(a\) in state \(s\), \(\max_{a^* \in A_{s'}} Q(s', a^*)\) is the maximum action value in the state that results from executing action \(a\) and \(\gamma\) is the discount factor parameter. The discount factor determines the effect of future rewards in calculating the action-value function for an action \(a\) and its value lies in the range of \([0-1]\). A value of 0 instructs the agent to consider only the current rewards when selecting an action, whereas a value approaching 1 indicates high importance being given to the action that leads to high rewards in future states.

3.2.3. Action selection

In our work, a simple action is a single interaction with a widget (Example: a button click) and a complex action is composed of multiple single actions (Example: enter text into multiple text fields on the current state of the AUT and then click a button). In our implementation of Q-learning, complex actions are created only for GUI states that contain text fields. We define a set of input values consisting of boundary, equivalence class, illegal and special values that are provided to the action selection component of Q-learning. These values are used to generate complex actions whenever the test generation system identifies text fields in a particular GUI state. This is because there is often a semantic relationship between the values entered in a text field and the subsequent non-text input actions such as a button click. The idea is to associate a Q-value to this set of semantically related simple actions. Treating each semantically related action as a separate entity does not provide a consistent measure of value because the behavior of a single action may vary depending on the nature of actions taken on other semantically related widgets. For instance, the behavior of an OK button on a screen is often dependent on the validation of values entered in the text fields.

We also use a simple variant of softmax action selection in our experiments instead of
always selecting the action with the highest Q-value in a particular state. In this strategy, each available action is weighted by its current Q-value and a weighted random selection is used to choose which action to execute. This action selection strategy introduces some randomness in the action selection process while still ensuring that the most valuable actions are most likely to be selected. The main aim of this strategy is to maintain some balance between exploitation and exploration of the AUT. The below equation shows how the selection probability of each action in a GUI state is calculated based on its Q-value, where $P(s, a)$ is the probability of selecting action $a$ in state $s$. $Q(s, a)$ is the current estimated Q-value for action $a$ and $\sum_{i=1}^{n} Q(s, a_i)$ is the sum of the current Q-values for all available actions in state $s$.

$$P(s, a) = \frac{Q(s, a)}{\sum_{i=1}^{n} Q(s, a_i)}$$

(3)

3.2.4. Algorithm

The algorithm for Q-learning based test suites generation is shown in Figure 3.6. The algorithm takes the configuration.json file and .apk file of the AUT as inputs from the user. The configuration file contains the details such as the value of discount factor parameter in the action-value function, the initial $r_{init}$ value for each available action, the initial action-value $Q$ for each available action, the $n$ number of test cases that need to be generated by the framework, etc.

The algorithm installs and starts the AUT in line 5. We consider the start of an episode, once the application starts. The variable testCase is initially set to empty. If the value generated for home_btn_prob in line 8 is less than or equal to 0.05, then the HOME button is chosen as the selectedAction in line 9, else, if the value generated for back_btn_prob is less than or equal to 0.05 in line 10, then the BACK button is chosen as the selectedAction in line 11, else, if the values of both home_btn_prob and back_btn_prob are greater than 0.05, the algorithm captures all the actions that are associated with the current state of the AUT by using the method getAvailableActions in line 13 and selects only one action, by using the method getActionByWeight in line 14. The getActionByWeight does a weighted random selection of an action from the set of actions in the current GUI state. The available actions are weighted by their current Q-values. The action selected is executed in line 15 and is appended to the current test case in line 16. If the selectedAction other
than `HOME` and `BACK` buttons execution exits the AUT or opens another app on the emulator, then the reward of the `selectedAction` is set to 0, as it does not contribute to explore or exploit the app, else, the actions available in the new resulting state are captured in line 23; the execution count of the `selectedAction` is increased by 1 in line 24; the new reward of the executed action is calculated and updated using the `Reward function` in line 25 and 26 respectively. We also calculate the new action-Value of the `selectedAction` and update the value as shown in the lines 27, 28 and 29. The lines 7 to 30 are repeated until the AUT exits or moves to the background. This generates a test case that is added to the test suite $T$. After each test case generation, the value of test cases count
is increased by 1 in line 32. The lines 4 to 33 repeat until the specified number of test cases are generated by the framework. The output of the algorithm is a test suite containing \( n \) test cases.

3.2.5. Example

Consider generating a test suite for an Android application \( A \). Let \( A \) have the states: \( a, b, c, d, e, f, g \) and \( h \) as shown in Figure 3.7. The possible actions in state \( a \) are \( a0 \) and \( a1 \). The possible actions in state \( b \) are \( b0, b1 \) and \( b2 \). The possible actions in state \( c \) are \( c0, c1 \) and \( c2 \). There are no actions available in the states \( d, e, f, g \) and \( h \). The directed arrow indicates transition from one state to another upon the execution of action that is mentioned above the arrow. (Example: Executing action \( a1 \) transits the state from \( a \) to \( c \))

Let the parameters associated with the function \( \text{generateTestSuite} \_Qlearning \) be: 1) \( r_{\text{init}} = 1 \); 2) \( \text{actionValue} = 1 \); and 3) \( \gamma = 0.5 \).

As per the algorithm shown in Figure 3.6, app \( A \) is started in line 5 and the main activity \( a \) is displayed to the user. In episode 0, line 13 of the algorithm sets the \( \text{currActions} \) value to \( \{a0, \)
Both \( a0 \) and \( a1 \) have the default \( r_{init} \) value as 1 and \( actionValue \) as 1. In line 14, weighted random action selection is performed on the \( currActions \), where an action with the largest weight is more likely to be selected. In our case, both the actions \( a0 \) and \( a1 \) have the same \( actionValue \). Hence, both \( a0 \) and \( a1 \) are equally likely to be selected. Assume that the action \( a0 \) is selected i.e. \( selectedAction \) is set to \( a0 \). In line 15, the \( selectedAction \) is executed. This will make a transition from state \( a \) to \( b \) and the value of \( newActions \) in line 23 will be \( \{ b0, b1, b2 \} \). Line 24 updates the execution count of action \( a0 \) to 1 (The execution count is 0 initially). Line 25 and 26 calculate and assign a new reward to action \( a0 \), whose value will be 3 here. In line 27, the maximum of \( \{ b0, b1, b2 \} \) is assigned to \( maxValue \), which is used in calculating new \( actionValue \) of \( a0 \) in line 28. In line 16, the \( selectedAction \) \( a0 \) is added to the \( testCase \). The lines from 7 to 30 repeat until an exit criterion is met by the algorithm.

Application exit indicates the end of an episode and a test case. The test case thus generated is added to the test suite \( T \). The algorithm repeats until the user specified number of test cases are generated by the framework. Below table gives the details of each action after each episode.

<table>
<thead>
<tr>
<th>State</th>
<th>Episode 0</th>
<th>Episode 1</th>
<th>Episode 2</th>
<th>Episode 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reward</td>
<td>Reward</td>
<td>Reward</td>
<td>Reward</td>
</tr>
<tr>
<td></td>
<td>Q-Value</td>
<td>Q-Value</td>
<td>Q-Value</td>
<td>Q-Value</td>
</tr>
<tr>
<td>a</td>
<td>a0</td>
<td>3</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.5</td>
<td>2</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>b0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>c0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>c1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 3.1.** Reward \( r_{init} \) and Action value \( actionValue \) captured for each action in each episode

Initially, at the beginning of episode 0, the actions \( a0 \) and \( a1 \) in state \( a \) have the same action-value. Suppose, the action \( a0 \) is chosen at random and executed, which transits the app state from \( a \) to \( b \). In state \( b \), the actions \( b0, b1 \) and \( b2 \) have the same action-value. Suppose, action \( b0 \) is chosen at random and executed, which transits the state from \( b \) to \( d \). No actions are associated with state \( d \). Hence, the application exits. As the execution of the action \( b0 \) is resulting in a state
that has no more actions to be explored, we set the reward and action-value of action $b0$ to 0. The $r_{init}$ and actionValue details of each executed action in episode 0 are calculated and updated in the Table 3.1 under the column episode 0.

In episode 1, the action $a0$ is chosen and executed, as it has largest weight of 3.5 and hence, the high probability of being selected. This transits the state from $a$ to $b$. In state $b$, the actions $b1$ and $b2$ have same values for actionValue. Suppose, $b1$ is chosen at random and executed. This transits the state from $b$ to $e$. As state $e$ has no actions associated with it, the application exits and episode 1 ends. The updated $r_{init}$ and actionValue of each executed action in episode 1 are given in Table 3.1 under the column episode 1.

Each episode is considered as a test case. Hence, the above algorithm repeats until $n$ test cases are generated. From Table 3.1 we find that our framework explores the action sequences \{a0, b0\}, \{a0, b1\}, \{a0, b2\}, \{a1, c0\} and \{a1, c1\}, without many repeated executions of the already executed actions. Hence, the Q-learning approach favors the exploration of newer actions and gaining a higher code coverage.
CHAPTER 4

EXPERIMENTAL SETUP

4.1. Applications under Test

Four Android apps from different domains are selected for our experiments. The apps are chosen from a repository of free and open source Android apps [2]. Table 4.1 gives the details such as the app name, version and number of code blocks, methods and classes in each AUT. The apps vary in size, which ranges from 4,959 to 22,169 blocks of code, where the number of methods and classes range from 227 to 782 and from 48 to 162 respectively. These apps contain a variety of input controls such as buttons, checkboxes, radio buttons, spinners, pickers, options menu, floating contextual menu, pop-up menu and dialog boxes. They also support the bytecode instrumentation. We instrumented our AUTs in order to track what code of the AUT has been executed during testing and to calculate the fine-grained code coverage details. Instrumentation is carried out using the byte code instrumentation technique that is described by Zhauniarovich et al. [36]. The instrumented .apk files are fed to our framework to generate code coverage reports along with the generation of tests.

<table>
<thead>
<tr>
<th>App</th>
<th>Version</th>
<th># Blocks</th>
<th># Methods</th>
<th># Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomdroid</td>
<td>0.7.2</td>
<td>22169</td>
<td>782</td>
<td>162</td>
</tr>
<tr>
<td>Droidshows</td>
<td>6.5</td>
<td>16244</td>
<td>467</td>
<td>75</td>
</tr>
<tr>
<td>Budget</td>
<td>4.0</td>
<td>9129</td>
<td>369</td>
<td>65</td>
</tr>
<tr>
<td>Moneybalance</td>
<td>1.0</td>
<td>4959</td>
<td>227</td>
<td>48</td>
</tr>
</tbody>
</table>

**Table 4.1.** Details of Android apps used in the experiment

4.2. Test Generation Parameters

The parameters involved in our framework such as the probabilities for selecting BACK and HOME buttons on the Android device, the initial reward value for each available action, and so on are configured manually in the `configuration.json` file. Table 4.2 shows the details of those parameters that are used in our experiments.

The probabilities for selecting BACK and HOME buttons determine the average length of test cases. These probabilities are set to 5% each, as they are noticed to provide a reasonable bal-
ance between short and long test cases. A time delay of 4 seconds is introduced in the framework, so that the AUT gets sufficient time to respond to the current action before performing the next action. The number of input text combinations is set to 10, which specifies that 10 text input combinations are generated for each GUI state that has two or more text input fields. Each of these combinations is tried out on the particular GUI state until a valid match is found and the state transition occurs. An initial reward value of 1 is assigned to all actions that are not explored yet, as the value of 1 ensures that valuable actions are repeated numerous times since they eventually gain higher Q-values. We chose the values 0.1, 0.3, 0.5, 0.7 and 0.9 for discount factor parameter and generated test suites using each discount factor value to evaluate the performance of Q-learning with varying discount factor parameter. The values 0.1 and 0.9 are chosen in order to test if the app’s code coverage improves when the rewards are given to the selected action either too late or very soon, where 0.1 corresponds to late reward and 0.9 corresponds to sooner reward. The discount factor value 0.5 is chosen to test if maintaining the balance between prioritizing immediate and future rewards improves the code coverage and the values 0.3 and 0.7 are chosen in order to test if the values between 0.1 - 0.5 or 0.5 - 0.7 improve the code coverage. The parameters number of text input combinations, discount factor and initial reward are not relevant for the UR approach.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>UR</th>
<th>Q-learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back button probability</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>Home button probability</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>Time delay between actions</td>
<td>4s</td>
<td>4s</td>
</tr>
<tr>
<td>Number of text input combinations</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>Discount factor</td>
<td>-</td>
<td>0.1, 0.3, 0.5, 0.7 and 0.9</td>
</tr>
<tr>
<td>Initial reward</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2. Tests generation parameters

4.3. Test Suites Generation

To reduce the effect of randomness in our experiments, UR and Q-learning approaches are run 10 times on each target app and the average results are used to evaluate their effectiveness. In the case of Q-learning approach, as we consider to evaluate the impact of the discount factor on the efficiency of Q-learning, 10 test suites are generated for each discount factor value that is given
in the Table 4.2. Each test suite is configured to contain 200 test cases.
CHAPTER 5

RESULTS AND DISCUSSION

In this chapter, we evaluate the effectiveness of Q-learning by measuring the average code coverage that is obtained for all the AUTs. The results of Q-learning are compared with that of the UR approach. We also discuss the average progressive block coverages achieved and the time taken for test suites generation using each approach.

5.1. Code coverage

Code coverage acts as a measure to find the exploration ability of an approach under evaluation. High code coverage indicates that the approach can explore a large part of the AUT. Effectiveness of Q-learning is investigated with different values for the parameter discount factor ($\gamma$). The parameter $\gamma$ influences the decision of what action to choose for execution. When $\gamma$ is approaching ‘1’, the decision of action selection favors the exploitation of the AUT, whereas, for $\gamma$ approaching ‘0’, the decision of action selection favors the exploration of the AUT.

In this section, we present the coverage results in terms of block, method and class coverages.

5.1.1. Block coverage

<table>
<thead>
<tr>
<th>App</th>
<th>UR</th>
<th>Q-learning Discount Factor 0.1</th>
<th>Q-learning Discount Factor 0.3</th>
<th>Q-learning Discount Factor 0.5</th>
<th>Q-learning Discount Factor 0.7</th>
<th>Q-learning Discount Factor 0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomdroid</td>
<td>48.13</td>
<td>46.74</td>
<td>48.52</td>
<td>53.15</td>
<td>53.55</td>
<td>49.18</td>
</tr>
<tr>
<td>Droidshows</td>
<td>59.86</td>
<td>58.09</td>
<td>58.20</td>
<td>66.18</td>
<td>59.48</td>
<td>49.31</td>
</tr>
<tr>
<td>Budget</td>
<td>72.96</td>
<td>73.57</td>
<td>72.70</td>
<td>74.23</td>
<td>72.71</td>
<td>69.48</td>
</tr>
<tr>
<td>Moneybalance</td>
<td>61.86</td>
<td>85.77</td>
<td>86.27</td>
<td>88.36</td>
<td>84.76</td>
<td>82.11</td>
</tr>
</tbody>
</table>

TABLE 5.1. Average block coverage achieved using UR and Q-learning algorithms

An app’s code contains multiple blocks of instructions. Block coverage indicates the percentage of code blocks that are explored while testing the app. Table 5.1 shows the average block coverages that are achieved in the experiments and the values in bold indicate the highest average
coverages obtained for that particular AUT. We observe that the Q-learning with a discount factor of 0.5 consistently achieved higher average block coverages than the UR approach for all the AUTs and performed better than the other Q-learning configurations for all the AUTs except Tomdroid. In regards to Tomdroid app, the discount factor 0.7 achieved a higher average block coverage. For the discount factors 0.1, 0.3, 0.7 and 0.9, the average block coverage is not consistently higher when compared with the UR approach. In addition to the average block coverages shown in Table 5.1, we also present other statistical details such as the least, highest and median block coverages that are achieved using each approach for each AUT in the following subsections.

5.1.1.1. Block coverages of Budget app

Block coverages obtained by UR and Q-learning approaches for the Budget app is presented as a box plot in Figure 5.1. Q-learning with 0.5 discount factor achieved the highest block coverage of 77.91% and the least coverage of 64.49% is recorded by Q-learning with 0.9 discount factor. The median coverages of all Q-learning configurations except 0.9 discount factor is higher in comparison with that of the UR approach.

5.1.1.2. Block coverages of Droidshows app

The details of block coverages for the Droidshows app are presented in Figure 5.2. Q-learning with 0.5 discount factor achieved the highest block coverage of 76.87% and the least
coverage of 13.14% is obtained by Q-learning with 0.9 discount factor. UR approach scored higher median block coverage when compared against the Q-learning with discount factors other than 0.5.

5.1.1.3. Block coverages of *Moneybalance* app

The block coverages of the *Moneybalance* app are shown in Figure 5.3. Q-learning with all discount factor configurations achieved higher median coverages and performed far better than UR approach. The highest coverage is 94.47% and the least coverage is 43.3%, which are obtained by Q-learning with 0.5 discount factor and UR approach respectively.

5.1.1.4. Block coverages of *Tomdroid* app

Figure 5.4 shows the block coverages of the *Tomdroid* app. Q-learning with 0.7 discount factor achieved the highest median coverage, whereas the 0.5 discount factor achieved the highest coverage of 61.21% and the least coverage of 38.61% is obtained by the Q-learning with 0.9 discount factor configuration.

5.1.1.5. Block coverages across all the AUTs

The box plot of block coverages achieved using the test suites that are generated by each approach across all the AUTs is shown in Figure 5.5. Q-learning with 0.5 discount factor has
the highest median block coverage and maximum coverage when compared with all the other Q-learning configurations and the UR approach. The Q-learning approach with discount factors other than 0.5 discount factor achieved higher median block coverages and higher block coverages when compared with the UR approach. The block coverage ranged from 43% (Tomdroid app) to 77% (Budget app) by using the UR approach. Q-learning with 0.5 discount factor covered from 48% (Tomdroid app) to 94.5% (Moneybalance app).
From the obtained block coverages, we observe that the highest block coverage and higher median block coverages across all the AUTs is achieved with Q-learning of 0.5 discount factor value. The performance of Q-learning with the other discount factor configurations is not consistently higher in comparison with the UR approach.

5.1.2. Method coverage

Method coverage indicates the percentage of an app’s code methods that are explored while testing the app. The average method coverages (in %) that are achieved by using UR and Q-learning approaches of our framework are shown in Table 5.2.

<table>
<thead>
<tr>
<th>App</th>
<th>UR</th>
<th>Q-learning Discount Factor 0.1</th>
<th>Q-learning Discount Factor 0.3</th>
<th>Q-learning Discount Factor 0.5</th>
<th>Q-learning Discount Factor 0.7</th>
<th>Q-learning Discount Factor 0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomdroid</td>
<td>51.37</td>
<td>50.41</td>
<td>50.92</td>
<td>54.64</td>
<td>55.17</td>
<td>51.88</td>
</tr>
<tr>
<td>Droidshows</td>
<td>65.89</td>
<td>66.04</td>
<td>65.83</td>
<td>73.71</td>
<td>67.45</td>
<td>56.27</td>
</tr>
<tr>
<td>Budget</td>
<td>80.87</td>
<td>81.11</td>
<td>80.7</td>
<td>81.41</td>
<td>79.54</td>
<td>77.16</td>
</tr>
<tr>
<td>Moneybalance</td>
<td>68.99</td>
<td>86.74</td>
<td>86.34</td>
<td>88.33</td>
<td>85.9</td>
<td>83.88</td>
</tr>
</tbody>
</table>

**Table 5.2.** Average method coverage achieved using UR and Q-learning algorithms
5.1.3. Class coverage

Class coverage indicates the percentage of an app’s code classes that are explored while testing the app. The average class coverages (in %) that are achieved by using UR and Q-learning approaches of our framework are shown in Table 5.3.

<table>
<thead>
<tr>
<th>App</th>
<th>UR</th>
<th>Q-learning Discount Factor 0.1</th>
<th>Q-learning Discount Factor 0.3</th>
<th>Q-learning Discount Factor 0.5</th>
<th>Q-learning Discount Factor 0.7</th>
<th>Q-learning Discount Factor 0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomdroid</td>
<td>62.78</td>
<td>61.73</td>
<td>62.28</td>
<td>66.05</td>
<td>65.19</td>
<td>61.98</td>
</tr>
<tr>
<td>Droidshows</td>
<td>68.93</td>
<td>66.93</td>
<td>68.13</td>
<td>74.53</td>
<td>67.34</td>
<td>58.93</td>
</tr>
<tr>
<td>Budget</td>
<td>85.08</td>
<td>84.92</td>
<td>84.46</td>
<td>85.23</td>
<td>82.92</td>
<td>80.31</td>
</tr>
<tr>
<td>Moneybalance</td>
<td>67.29</td>
<td>76.25</td>
<td>76.25</td>
<td>76.87</td>
<td>75.62</td>
<td>74.79</td>
</tr>
</tbody>
</table>

**Table 5.3.** Average class coverage achieved using UR and Q-learning algorithms

From Tables 5.2 and 5.3, we notice that, in the case of method coverage also, Q-learning with 0.5 discount factor consistently performed better than the UR approach for all the AUTs. The performance is also higher when compared with the other Q-learning configurations for all the AUTs except Tomdroid. In regards to class coverage, Q-learning with 0.5 discount factor consistently performed better than the UR approach and all the other Q-learning configurations for all the AUTs.

In the next section, we compare the average progressive block coverage results of UR and Q-learning with 0.5 discount factor configuration. As blocks are the primary and basic components of an app’s code, we are considering only block coverage reports in comparing the progressive coverages.

5.2. Progressive Block Coverages

Figures 5.6, 5.7, 5.8 and 5.9 show how our framework incrementally increases the average block coverage for **Budget**, **Droidshows**, **Moneybalance** and **Tomdroid** apps respectively across succeeding sets of 25 test cases.

For all the apps excluding **Moneybalance**, though the UR approach initially performed better than the Q-learning, as the number of tests cases increased, the coverage obtained by Q-learning exceeded that of the UR approach. The reason is that the UR approach generates the same set of
Figure 5.6. Progressive block coverages of the app *Budget*

Figure 5.7. Progressive block coverages of the app *Droidshows*

redundant actions leading to a fall in its exploration ability. Hence, the coverages gained by UR approach increase only slightly as the number of test cases increase. The Q-learning finds the actions based on their reward and Q-value, which eventually tries to select and execute the previously
unexecuted or lesser executed actions, thus aiming for a higher code coverage. For *Budget* app, the coverage achieved by the Q-learning and UR approaches is almost the same. For *Tomdroid* app, the initial coverages are almost the same at the end of first 50 test cases. However, Q-learning begins to attain higher code coverage after that. In the case of *Droidshows* app, the UR approach initially performed better for the first 50 test cases. But, later the coverage of the Q-learning approach is improved over UR approach. The most significant difference in progressive coverage was observed for *Moneybalance* app, where the Q-learning achieved much higher coverage at 25 test cases and continued to maintain the higher coverage throughout the test suite. The main reason is that the *Moneybalance* app primarily contains GUI states that have multiple text input fields. For the GUI states containing text input fields, the Q-learning approach generates complex actions in order to assign meaningful Q-values, which help the Q-learning approach perform better than UR approach.

5.3. Test suites generation time

The average time taken to generate a test suite using UR and Q-learning approaches for each subject application is presented in Table 5.4. The number of text input fields present in a GUI state of the app affects the amount of time taken to generate a test suite. For a text field, the
text entry action consumes a considerable amount of time. The more the number of text fields in a particular GUI state, the number of text validations grow and the larger is the time taken to perform a valid action, as the framework tries out all the text combinations until a valid text combination is entered. This increases the time taken to generate a test case. The Droidshows app has only two GUI states that have a maximum of 1 text field; the Tomdroid app has six GUI states that have a maximum of 2 text fields; the Budget app has one GUI state that contains 3 or more text input fields and the Moneybalance app has three GUI states that contain 3 or more text input fields. Q-learning puts effort in composing complex actions if there are any text fields in a GUI state. Hence, the time taken by Q-learning is slightly higher than the UR approach for Tomdroid, considerably high in the case of Budget and double in the case of Moneybalance.

<table>
<thead>
<tr>
<th>App</th>
<th>UR</th>
<th>Q-learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomdroid</td>
<td>6.41</td>
<td>6.89</td>
</tr>
<tr>
<td>Droidshows</td>
<td>6.53</td>
<td>6.16</td>
</tr>
<tr>
<td>Budget</td>
<td>12.57</td>
<td>14.94</td>
</tr>
<tr>
<td>Moneybalance</td>
<td>7.03</td>
<td>14.02</td>
</tr>
</tbody>
</table>

TABLE 5.4. Average tests generation time (in hours)
5.4. Implications for Software Testers

From section 5.1 and 5.2, we observe that the Q-learning with 0.5 discount factor achieved higher median block coverages and the maximum block coverages for all the AUTs. The average block coverages are also maximum for all the AUTs except Tomdroid with Q-learning of 0.5 discount factor. As the discount factor value approaching '0' usually concentrates on exploring the app, the nested or deeper AUT paths are unexplored, resulting in a lower code coverage. In the case of discount factor value approaching '1', as the AUT is exploited, the initially chosen AUT paths are traversed in a deeper or nested fashion, resulting in a lesser exploration of the other paths. Hence, the discount factor approaching 1 also results in a lower code coverage. This pattern is observed clearly in the case of Droidshows and Moneybalance apps as shown in Table 5.1. However, the data for Tomdroid app shows that the 0.7 discount factor performed better than the other discount factor configurations and the block coverages decreased gradually as the discount factor is moved towards '1' and '0' from the value 0.7. In the case of Budget app, moving the discount factor from 0.5 towards 1 shows the fall in the average block coverage, but the coverage increased at 0.1 discount factor. Hence, we observe that along with the value of discount factor, the AUT's execution space also influences the coverage achieved. However, in our research we target to achieve a higher and consistent coverage without any knowledge of the execution space of the AUT. Our experimental data suggests that better results are obtained by maintaining the balance between prioritizing exploration and exploitation. Hence, we recommend a value of 0.5 for the discount factor parameter.
Automated GUI test suites generation for Android apps is a challenging task because it involves the simulation of user actions on the Android devices and generation of a sequence of actions dynamically by interacting with the app in a meaningful manner. It is also time-consuming due to the AUT’s large execution space that needs to be explored and requires significant human effort for writing the tests. Many Android app testing tools such as capture-and-replay tools and libraries for writing the test scripts are available in the market for automated GUI tests generation, but these tools still require a considerable amount of time and human effort. Hence, in this work, we present and implement Q-learning, a reinforcement learning algorithm that automates the GUI tests generation for Android apps in a dynamic fashion by reducing the human effort and time required to analyze the possible test cases and write the tests. Using the concept of dynamic action selection, Q-learning algorithm makes an intelligent selection of a GUI action from the available set of GUI actions. The novel part of this approach is that it does not require any previously constructed abstract model of the AUT.

We evaluated the performance of Q-learning approach against the UR approach, a basic method for generating tests at random. The performance is evaluated in terms of the code coverage achieved by each proposed approach. The results show that given the same test generation parameters and a specified number of test cases to generate, the Q-learning approach achieves a higher average code coverage than the UR approach. Hence, the choice of Q-learning, in the place of random algorithms will yield efficient test cases that achieve better code coverage for the given AUT.

In our current work, several parameters such as the reward function, discount factor values, initial reward values, initial Q-values, etc. are involved. The selection of values for these parameters will affect the behavior of the Q-learning algorithm. In our current implementation, we evaluated Q-learning with different configurations of the discount factor parameter. The results indicate a better code coverage with a value of 0.5 for the discount factor. Further experimentation
is needed with varying values of other configuration parameters. As a part of the future work, we also plan conducting experiments on more Android apps and generating more test suites for each app. This might help increase the confidence in the current results and to evaluate the best choice of values for the associated parameters.
APPENDIX

ARCHITECTURE OF TESTS GENERATION FRAMEWORK
The architecture of our test suites generation framework is divided into four modules as shown in the Figure A.2. The details of each module are discussed below.

A.1. Input

The input to our framework comprises the following:

- Configuration file
- Android application package

a. Configuration file

Several parameters are associated with our framework. We created a `configuration.json` file to maintain all the necessary parameters in terms of key-value pairs. These parameters are configured by the user before the start of framework execution and provided as an input to the framework. The sample configuration file is shown in the Figure A.1 and few important parameters are discussed below:

- `aut_dir` specifies the directory path in which the `.apk` file of the AUT is placed.
- `aut_app` specifies the `.apk` file that needs to be chosen from the set of available `.apk` files in the `aut_dir`.
- `aut_package` defines the package name of the AUT. The test suite generated by our framework is stored under this package name.
- `number_of_testcases` specifies the number of test cases that need to be generated by the framework as a test suite.
- `home_btn_prob` defines the probability of selecting HOME button on the emulator during the process of event selection.
- `back_btn_prob` defines the probability of selecting BACK button on the app’s UI during the process of event selection.
- `discount_factor` specifies the value of $\gamma$. 
Figure A.1. Sample configuration.json file

- `text_reward_init` defines the initial default reward value for text field entries.
- `nontext_reward_init` defines the initial default reward value for non-text fields. The parameters `discount_factor`, `text_reward_init` and `nontext_reward_init` are associated only during the process of test suites generation using Q-learning approach.

b. Android application package

Android Application Package (APK) is the file format that is used for installation of the Android applications. It is an archive file that contains program’s code (.dex files), resources, certificates, and manifest file. The .apk file of the AUT must be given as an input to our framework that installs and launches the AUT on the emulator.

A.2. Tests generator

The tests generator module implements the UR and Q-learning approaches, and targets the generation of test cases, coverage files and log files. For each test case, a corresponding coverage file and log file are generated. The coverage file holds the information regarding the code coverage that is obtained by that test case and the log file contains the android system level logs that are generated during that test case generation. Any exceptions and errors that occur during this process are captured in the log files, that are further analyzed in finding the bugs that are associated with
This module launches the AUT on the Android emulator; obtains the set of actions that are available on the current activity of the AUT; makes an action selection from the set of available actions based upon the user-specified algorithm and executes the selected action on the AUT that is running on the emulator. If the algorithm specified is random, the framework will select actions based on the UR approach. Otherwise, if the algorithm is given as Qlearning, the framework will select actions based on the Q-learning approach. The tests generator module executes until the user-specified number of test cases are generated by the framework.

A.3. Android Emulator

We use Genymotion, an Android emulator in our research [3]. It comprises a complete set of sensors and features that enable the interaction with virtual Android environment. It requires on Oracle VM Virtual Box and drivers for its working. However, Genymotion is simple to install, easy-to-use and fast. It enables the users to run and test their Android apps on a wide range of virtual Android devices. Our tests generator module starts the AUT on Genymotion. Once started,
the AUT running on Genymotion responds to the actions that are fired by the tests generator module and transits the app from one state to another based on the action fired.

A.4. Output

The output of our framework consists of the following:

- Test suite
- Coverage files
- Log files

a) Test Suite

The framework generates `number_of_testcases` (a parameter that is configured in `configuration.json` file) test cases as a single test suite. Each test case contains a sequence of actions that are chosen and executed to test the AUT. The test cases are saved in the format `TestcaseXXXXX.json`, where `XXXXX` specifies the test case number. For example, if the test case generated is the 126\(^{th}\), then it is saved as `Testcase00126.json`.

b) Coverage Files

Each test case generation is followed by the generation of a coverage file. The coverage files contain fine-grained code coverage details. These files are saved in the format `coverageXXXXX.ec`, where `XXXXX` specifies the corresponding test case number. We developed a Python script that takes these coverage files as input and calculates the block, method and class coverage results.

c) Log Files

Each test case generation is followed by the generation of a log file. The log files are saved in the format `logXXXXX.txt`, where `XXXXX` specifies the corresponding test case number. These are system level log files that are generated on Android emulator during the action execution on the emulator. These log files are used to find if any errors or exceptions occurred during the tests generation.


