Journal of Computer Science

Original Research Paper

Improving the Exploration Strategy of an Automated Android GUI Testing Tool based on the Q-Learning Algorithm by Selecting Potential Actions

Goh Kwang Yi, Salmi Binti Baharom and Jamilah Din

Department of Software Engineering and Information System, Universiti Putra Malaysia, Malaysia

Article history
Received: 29-07-2021
Revised: 02-11-2021
Accepted: 15-01-2022

Corresponding Author:
Salmi Binti Baharom
Faculty of Computer Science and Information Technology,
Universiti Putra Malaysia
43400, Serdang, Selangor
Darul Ehsan, Malaysia
Email: salmi@upm.edu.my

Abstract: Researchers have proposed automated testing tools to minimise the effort and resources spent on testing GUIs. A relatively simple strategy employed by the proposed tools thus far is the *observe-select-execute* approach, where all of a GUI's actions on its current state are observed, one action is selected and the selected action is executed on the software. The strategy's key function is to select an action that may achieve new and desirable GUI states. Due to difficulties in comparing actions, most existing test generators ignore this step and randomly select an action. However, a randomly selected action has limitations. It does not test most parts of a GUI within a reasonable amount of time and there is a high probability that the same actions are re-selected. This reduces code coverage, thereby resulting in undetected failures. To overcome this limitation, the Q-Learning algorithm was proposed by several researchers to minimise randomness. The idea was to change the probability distribution over the sequence space. Instead of making purely random selections, the least frequently executed action is selected so that the GUI can be further explored. Q-Learning showed better results than the random exploration strategy but it also presented a weakness. Q-Learning's reward function assigns the highest value to the least frequently executed action without taking into consideration its potential ability in detecting failures. Furthermore, the proposed techniques based on the Q-Learning algorithm do not consider context-based actions. Thus, these techniques are unable to detect failures that occur due to the improper use of context data, which is becoming an increasingly common issue in mobile applications nowadays. We propose a tool, namely the Crash Droid, that allows the automation of testing context-aware Android applications. We utilise the Q-Learning algorithm to compare actions, including context-based actions, to effectively detect crashes and achieve a higher code coverage.

Keywords: Automated Mobile GUI Testing, Q-Learning, Context-Aware

Introduction

Smartphones have become a crucial part of our lifestyles. Mobile applications have transformed the way we perform daily activities, whether it's ordering food, booking a flight, paying bills or and chatting with friends. Considering the fact that 3.2 billion smartphones were sold, 8.3 billion mobile subscriptions were registered, more than 3.14 million applications were developed and 204 billion applications were downloaded worldwide in 2019 (Statista.com, 2021d; 2021b; 2021c; 2021a), the significance of testing should not be neglected for quality assurance purposes. The quality of mobile applications is a key factor in determining user satisfaction. Poor usability would frustrate users and prompt them to uninstall an application. Frozen screens, crashes, unresponsiveness and high battery consumption can contribute to this frustration (Inukollu *et al*., 2014). Testing mobile applications is an expensive, time consuming and challenging task. One of the reasons is that mobile applications require frequent updates to improve user experience, fix bugs and compete for users' attention. Frequently releasing updates shortens its development time, thereby making it harder to ensure the quality of mobile applications due to insufficient testing. Thus, researchers have suggested automation as a solution to

Science Publications

© 2022 Goh Kwang Yi, Salmi Binti Baharom and Jamilah Din. This open access article is distributed under a Creative Commons Attribution (CC-BY) 4.0 license.

accelerate its development, in particular its testing process. Another reason is that context-aware applications are becoming increasingly common in mobile applications. Context-aware applications are difficult to test because changes in context data can affect software behaviour at any point during execution. Furthermore, context data is generally inaccurate, inconsistent and continuous, making the applications even more challenging to test than those without context data (Yue *et al*., 2016).

Mobile applications are highly dependent on their Graphical User Interfaces (GUIs) due to their event-driven nature and gesture-based interaction. For this reason, GUI testing often replaces system testing. Testing GUIs involves creating sequences of GUI events that exercise GUI widgets (i.e., test cases), executing those events (i.e., test execution) and monitoring resulting changes to the software state (i.e., test oracle) (Memon *et al*., 2003; Nguyen *et al*., 2014) Even though the creation of test cases is associated with GUI widgets, research has shown that GUI testing is effective at finding both GUI and non-GUI faults (Robinson and Brooks, 2009). This is because the test cases do not only execute GUI codes but may also execute non-GUI codes. GUI testing can be used to identify security flaws, crashes and exceptions that occur while using mobile applications. All these require the simulation of user actions on the software and therefore automatic GUI testing needs to mimic human interaction with the GUI widgets.

A relatively simple strategy used in automated GUI testing tools is the *observe-select-execute* approach. The strategy starts by launching the Application Under Test (AUT) and then proceeds by observing the GUI actions on the AUT's current state, selecting an action from those observed actions and executing the selected action. The strategy's key function is to select an action that may achieve new and desirable GUI states. Due to difficulties in comparing actions, most existing tools ignore this step and randomly select an action. However, for large GUIs with numerous and deeply nested actions, a random algorithm is unable to sufficiently test most of its parts within a reasonable amount of time. Furthermore, it does not explore the AUT systematically. Since the actions are chosen at random, there is a high chance that previously selected actions are selected again, resulting in lower code coverage and unrevealed failures.

To overcome the limitations of the random algorithm, several researchers (Bauersfeld and Vos, 2012; Buzdalov and Buzdalova, 2013; Carino and Andrews, 2016; Koroglu and Sen, 2018; Mariani *et al*., 2012) have proposed Q-Learning to improve the probability distribution over the sequence space by exploiting a learning engine. Instead of randomly selecting an action, the least frequently executed action is selected so that the GUI can be more thoroughly explored to maximise coverage and locate crashes. The prospect of discovery in such an approach is considered more "interesting" to a tester. However, these techniques select an action based solely on its frequency of execution without taking into consideration its potential ability in detecting and revealing failures. For example, let's compare the actions of tapping a button to submit data to a database and tapping a button to reset data within the interface. If both these tapping actions have never been executed, the probability of each action to be selected would be equal if the selection is based solely on the frequency of execution. However, the former button executes a complex code where it might involve data transmission over the network and multiple servers. Hence, from a tester's point of view, the action has a greater potential for bringing more interesting results than the latter. Unfortunately, the action's ability to detect crashes is not considered. Furthermore, these techniques do not consider context-aware applications, therefore they may not detect defects that occur due to the improper use of context data.

This is an ongoing research that aim to propose a testing tool that is able to automatically test Android applications. The Android platform is selected as it is the most popular mobile operating system in the world. As of July 2017, the number of available applications available on Google Play Store is 2.95 billion (Statista.com, 2019). Its popularity among developers is owing to the accessible development environment that is based on the familiar Java programming language as well as the availability of open-source libraries implementing diverse functionalities that accelerate the development process.

We name our proposed tool Crash Droid. It is based on the *observe-select-execute* strategy and utilises that Q-Learning algorithm to compare actions. However, we enhanced Q-Learning's function by adding the ability to compare context-based actions as well, so as to improve its competency at exploring a GUI in order to effectively detect crashes and achieve a higher code coverage. In this study, the conceptual design of our proposed tool, Crash Droid that addresses the issue of improving the exploration strategy of the Q-Learning algorithm by selecting actions, including context-based actions, based on their potential abilities in uncovering failures.

The Conceptual Design of the Tool

We propose a tool named Crash Droid that interacts with and explores an AUT using the observe-select-execute strategy, where all the possible GUI actions on the AUT's current state are observed, one action is selected based on its crash detection potential and the selected action is executed on the AUT. The tool employs the Q-Learning algorithm with the purpose of further exploring the GUI to maximise coverage and locate crashes. Figure 1 shows an overview of Crash Droid that consists of two phases, which are (1) the pre-testing phase and (2) the testing phase. The details of these phases are discussed below.

The Pre-Testing Phase

Pre-testing addresses the issue that every action was previously treated to have the same potential. In this study, actions are differentiated by its weight, calculated before testing takes place. The weights are used as the basis in determining the initial action value, which could potentially speed up the crash of an AUT. The weight is calculated based on two categories of metrics: (1) Non-context-aware and (2) context-aware. These metrics are extracted from the action's underlying code. The non-context-aware metric is related to the complexity of the code and the number of called functions, also known as the Response For Call (RFC). The complexity of the code is calculated using the cyclomatic complexity formula proposed by McCabe. The context-aware metric is related to the network and GPS used by the code. They can be calculated using their related functions, such as:

for GPS:

- getMaxSatellites ()
- getSatellites ()
- requestLocationUpdates ()

for network:

- httpClient.execute (httpPost)
- httpResponse.getEntity ()

The process of calculating the weight is discussed below.

Weight Calculation Process

Consider the actions and their corresponding values for the metrics Response for Call (RFC), Complexity of Code (CC), number of network-related functions (Net) and number of GPS-related functions (GPS). Table 1 shows the metric values of actions in a sample application.

The Response for Call (RFC) Weight of an Action

The RFC weight of an action A_i is expressed as the number of functions called by its code relative to the highest number of functions called by an action in the AUT. The RFC weight, $W_{rfc}A_i$, is computed by dividing the number of functions called by the code in A_i by the highest number of functions called by an action in the AUT. So, if action A_i calls N_{rfc} functions and the highest number of functions called by an action in the AUT is M_{rfc}, then $W_{rfc}A_i$ is calculated as:

$$W_{rfcA_i} = N_{rfc} \div M_{rfc} \quad (1)$$

where:

N_{rfc} - number of functions called by the action A_i

M_{rfc} - highest number of functions called by an action in the AUT

The weight W_{rfc} a0 of action a0 in Table 1 is 0.8750 as shown in Table 2.

The Cyclomatic Complexity (CC) Weight of an Action

The cyclomatic complexity metric determines the complexity of a code, The cyclomatic complexity weight of an action A_i is expressed as the complexity of its code relative to the highest complexity among all actions in the application. The weight, $W_{cc}A_i$, is computed by dividing the complexity of action A_i by the highest complexity of an action in the AUT.

$$W_{rfcA_i} = N_{cc} \div M_{cc} \quad (2)$$

where:

N_{cc} is the complexity of action A_i
M_{cc} is the highest complexity of an action in the AUT

The weight $W_{cc}a0$ of action a0 in Table 1 is 0.3750 as shown in Table 2.

The Network-Related Function Weight of an Action

The network-related function metric, which is a context metric, determines the network used in a code. The network-related function weight of an action A_i is expressed as the number of network-related functions in its code relative to the highest number of network-related functions in the code of an action in the AUT. The weight $W_{net}A_i$ is computed by dividing the number of network-related functions in action A_i by the highest number of network-related functions in an action in the AUT.

$$W_{net}Ai = N_{net} \div M_{net} \quad (3)$$

where:

N_{net} = The number of network-related functions in action A_i
M_{net} = The highest number of network-related functions in an action in the AUT

The weight $W_{net}\,a_0$ of action a_0 in Table 1 is 0 as shown in Table 2.

The GPS-Related Function Weight of an Action

The GPS-related function metric determines the use of GPS in a code. The GPS-related function weight of an action A_i is expressed as the number of GPS-related

DOI: 10.3844/jcssp.2022.90.102

functions in its code relative to the highest number of GPS-related functions in the code of an action in the AUT. The weight $W_{gps}A_i$ is computed by dividing the number of GPS-related functions in action A_i by the highest number of GPS-related functions in an action in the AUT.

$$W_{gps}Ai = N_{gps} \div M_{gps} \tag{4}$$

where:

N_{gps} = The number of GPS-related functions in action A_i
M_{gps} = The highest number of GPS-related functions in an action in the AUT

The weight $W_{gps}a_0$ of action a_0 in Table 1 is 0 as shown in Table 2.

The weights of each action in the sample application given in Table 1 for the metrics RFC, Cyclomatic Complexity, number of related Network-related functions and number of GPS-related functions are presented in Table 2.

For m metrics, the total weight AW_{ij} of an action A_i is computed as follows:

$$AW_{ij} = \frac{\sum_{j=1}^{m} Wx_j A_i}{m} \tag{5}$$

where:
x_j = The action metric
m = The total number of metrics

The maximum total weight of an action is 1, meaning it is the action with the highest metric value in the AUT. A weight of 0 means that the metric is not applicable in this action.

The calculated weight values of actions in the sample application for each of their corresponding metrics and the total weight of each action are given in Table 2.

The Testing Phase

The automated testing of an AUT takes into consideration both non-context-aware and context-aware applications. Crash Droid employs the Q-Learning algorithm to select the action with the highest crash-detection potential. If the selected action is context-aware, the environment requires some adjustment prior to its execution in order to test the context-aware attribute. The Q-Learning algorithm is a model-free reinforcement learning technique. In Q-Learning, an agent goes through numerous trials of interactions with a complex and uncertain environment. The agent learns the optimal action-selection procedure through those interactions to find the best action that would produce the desired state. The ultimate goal of the learning process in the long run is to maximise the total reward from every successive interaction with an AUT. In the context of our work, Crash Droid plays the role of the agent that goes through the trial-and-error interactions with an AUT (i.e., the environment) with the intention of causing the AUT to crash. The agent starts with limited knowledge about an AUT. Then, through the exploration and exploitation of the AUT, the agent learns and gains more knowledge. A selected action is assigned a value that is determined by the Q-value function, Q. The process of selecting the best action is defined by the value of Q. Upon executing the action, the agent is awarded with a reward that is determined using the reward function R. The definitions of the Q-value function Q and the reward function R are described below.

Reward Function

The reward function calculates the reward value of an action that transforms an AUT's current state into a new state. The reward function is defined to enable the agent to compare the crash detection potentials of actions. A higher reward value is awarded to actions with more potential than those with less potential. We define the reward function R for taking action α in state s of an AUT that leads to state s' as follows:

$$R(s,a,s') = \begin{cases} \gamma init' & if\ xa = 0 \\ \frac{1}{xa} \times a_{s'}, & otherwise \end{cases} \tag{6}$$

where:
γ_{init} = The initial default reward
x_α = The number of times action α has been executed in state s
α_s = The number of actions in state s that were not in state s

In this study, the initial default reward uses a value that is different from those used in other existing studies. Instead of a constant value, as used by many researchers, we use the total weight value plus one. The objective is to guide the selection of actions from as early as the first interaction with the AUT to speed up crash detection. The reward for subsequent activities is awarded based on actions that were less-frequently selected during the testing process.

The more frequently an action is executed, resulting in a new state with fewer actions, the less appealing it would be to the agent. The reward function in this study will explore actions with greater potential and were less-frequently explored in the past as they hold high reward values. The frequency criterion ensures that an action is not selected repeatedly when exploring an AUT.

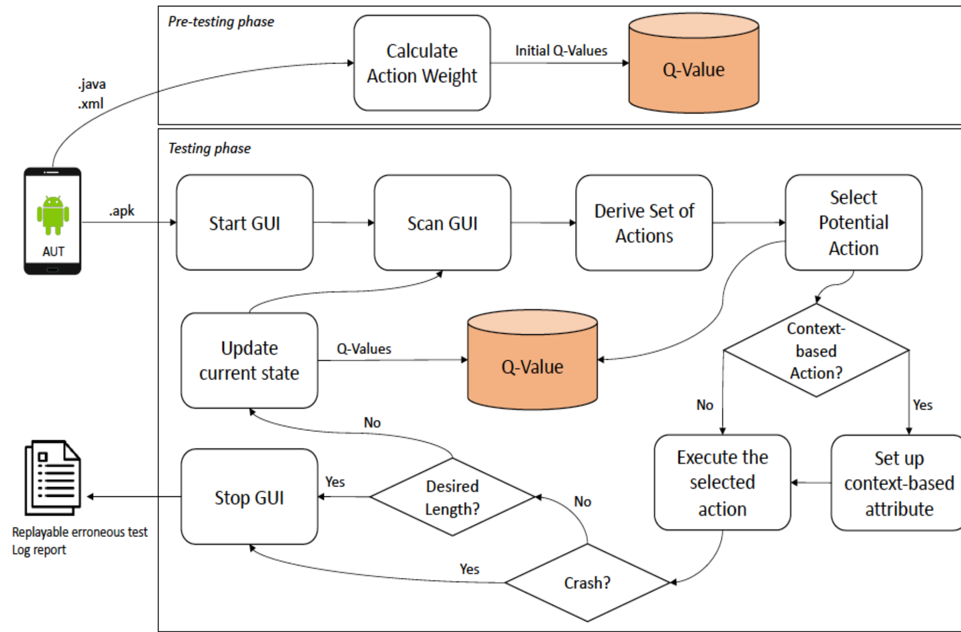

Fig. 1: Overview of Crash Droid

Table 1: Actions and their corresponding metric values

Action	RFC	CC	Net	GPS
a0	7	3	0	0
a1	6	1	0	0
b0	7	4	0	0
b1	4	8	0	0
b2	8	1	0	0
c0	6	1	0	0
c1	4	2	0	0

Table 2: Actions and their corresponding metric weights

Actions	RFC weight	CC weight	Net weight	GPS weight	Total weight
a0	0.8750	0.3750	0	0	0.3125
a1	0.7500	0.1250	0	0	0.2188
b0	0.8750	0.5000	0	0	0.3438
b1	0.5000	1.0000	0	0	0.3750
b2	1.0000	0.1250	0	0	0.2813
c0	0.7500	0.1250	0	0	0.2188
c1	0.5000	0.2500	0	0	0.1875

Table 3: Total weights and initial q-values of actions

Action	RFC weight	CC weight	Net weight	GPS weight	Total weight	Initial Q-value
a0	1	1	0.5	1	0.8750	1.8750
b0	0.6	0.5	1	0	0.5250	1.5250
b1	0.6	0.5	1	0	0.5250	1.5250
b2	0.2	0.5	0	0	0.1750	1.1750

Q-Value Function

The Q-value function, Q calculates the value of an action α, which is present in a particular state s of an AUT. It uses the value of the immediate reward for executing action α and the optimal future reward associated with action α. This function is crucial because it allows the agent to plan ahead when deciding what action to select in a particular state. The Q-value function is defined as follows:

$$Q(s,a) = R(s,a,s') + \gamma . \max a^{\cdot} \in As' Q(s',a^*) \qquad (7)$$

where:

$Q(s, a)$ = The value of action $^{\alpha}$ that is present in state s

$R(s, a, s)$ = The reward value for executing action α in state s

$max_{\alpha^*} \in A_s\, Q(s, a^*)$ = The maximum action value in the state that results from executing action α; and

γ is the discount factor parameter The discount factor determines the effect of future rewards in calculating the Q-value function for an action α and its value lies within the range of [0-1]. A value of 0 instructs the agent to consider only the current reward when selecting an action, whereas a value approaching 1 indicates high importance being given to an action that leads to high rewards in future states.

The Jaccard Distance

In general, when observing actions in the testing phase, the action that carries the highest weight will be selected. However, there is a possibility that several actions might have the same weight value. In this study, the Jaccard distance is employed to prevent the tool from randomly selecting an action among those of the same weight. The Jaccard Distance is used to compare similarities and diversities between sample sets. We use the Jaccard Distance to measure similarities between actions using the four metrics. It is calculated using the following Eq. 8:

$$\text{Jaccard Distance } (P_a, P_b) = 1 - \frac{\left|P_a \bigcap P_b\right|}{\left|P_a \bigcup P_b\right|} \quad (8)$$

where, P_a and P_b represent actions that consist of different sets of metrics. The value of the Jaccard Distance may vary between 0 and 1. A distance value of zero means that both actions are the same. A distance value of 1 indicates that there is no similarity between the two actions.

Jaccard Distance Calculation Process

Consider testing an AUT that has five states as in Fig. 6. The states are represented as circles. The transition from one state to another upon the execution of an action is shown by the arrows. Based on the figure, executing action a0 transitions the state from a to b. The possible actions in state b are b0, b1 and b2 and there are no available actions in states *d*, *e* and *f*.

The total weight is calculated as described earlier in the Weight Calculation Process subsection. The initial Q-value is calculated by adding 1 to the total weight. The possible actions to select in state *b* are b_0, b_1 and b_2. From the table, the highest Q-value among them is 1.5250. However, the agent is unable to select an action because there are two actions with that value, which are b_0 and b_1. To avoid randomly selecting b_0 or b_1, the Jaccard Distance is used to determine which of the two actions has a better potential. In order to decide whether to choose b_0 or b_1, we calculate the similarity score between a_0 and b_0 as well as the similarity between a_0 and b_1 using the Jaccard Distance. Then we compare the two similarity scores and select the action with the highest score.

First, we calculate the similarity score between a_0 and b_0. Figure 7 shows the codes and the corresponding metrics for a_0 and b_0. Based on the given information, we calculate the similarity score for each metric and obtain the average score.

Similarity Score for RFC

A total of five and three functions are recorded for a0 and b0 respectively. Two of the functions are used in both a0 and b0. Thus, the Jaccard Distance (a_0, b_0) for RFC = 1 - (2/6) = 0.67.

Similarity Score for CC

A total of three and one conditions are recorded for a0 and b0 respectively. They share no common conditions. Thus, the Jaccard Distance (a_0, b_0) for CC = 1 - (0/4) = 1.

Similarity Score for Net

A total of one and two network-related functions are recorded for a0 and b0 respectively. One of the functions is used in both a_0 and b_0. Thus, the Jaccard Distance (a_0, b_0) for Net = 1 - (1/2) = 0.5.

Similarity Score for GPS

A total of two GPS-related functions is recorded for a_0 and no function is recorded for b_0. Thus, the Jaccard Distance (a_0, b_0) for GPS = 1 - (0 /2) = 1.

Average Similarity Score

Average Jaccard Distance (a0, b0) = (0.67 + 1 + 0.5 + 1) / 4 = 0.7925.

Next, we calculate the similarity score between a_0 and b_1. Figure 8 shows the codes and the corresponding metrics for a_0 and b_1.

The similarity scores are 0.67, 0.67, 0.5 and 1 for RFC, CC, Net and GPS respectively. The average similarity score between a_0 and b_1 is 0.71. The two average similarity scores (i.e., 0.7925 and 0.71) indicate that a_0 is not similar to either b_0 or b_1. However, b_0 has a higher similarity score to a_0 than b_1, thereby indicating that b_0 has a higher potential for crash detection. Therefore, in this case b_0 is selected by the agent.

Example

Consider an Android application that has states A, B, C, D, E, F, G and H as shown in Fig. 9. State A has two

DOI: 10.3844/jcssp.2022.90.102

possible actions (a_0 and a_1), state B has three (b_0, b_1 and b_2) and state C has two (c_0 and c_1). States D, E, F, G and H are leaf nodes that represent the results of executing the termination actions. The transitions from one state to another upon execution of the actions are shown by the arrows. Based on the figure, executing action $a0$ transitions the state from A to B. The possible actions in state B are $b0$, $b1$ and $b2$, the possible actions in state C are $c0$ and $c1$ and there are no possible actions in states D, E, F, G and H.

Instead of using a constant value, our approach calculates the initial Q-value by adding 1 to the total weight. The calculation of the total weight is described in the Weight Calculation Process subsection. The initial Q-values for the actions in Fig. 9 are shown in Table 4.

We illustrate the testing phase of Crash Droid based on the information given in Fig. 9 and Table 4. At the start of episode 0, which is the beginning of the testing phase, the agent is in state A. State A has two possible actions, which are $a0$ and $a1$. Since $a0$ holds the higher Q-value, the agent selects and executes $a0$. This causes a transition from state A to state B. Since B is not a terminal state, the reward for executing $a0$ is calculated as defined in Eq. 6 and a new Q-value for $a0$ is calculated as defined in Eq. 7. The agent then continues in State B that has three actions, which are $b0$, $b1$ and $b2$. Since $b1$ holds the highest Q-value, the tool selects and executes $b1$. This causes a transition from state B to state E. The reward and Q-value for $b1$ are set to 0 since state E is a terminal state. The agent repeats this process for each episode until it executes a termination action that closes the application. Table 5 shows the reward and Q-value for each action after each episode.

Empirical Evaluation

We conducted an experiment to investigate the significance of the difference in the potential abilities of actions when testing Android applications by comparing the percentages of code coverage achieved by the approaches under comparison. The goal of the experiment is to answer the question, "Is the Crash Droid more effective than the approach under comparison?". We compared our approach with another approach proposed by Adamo *et al.* (2018), namely the Auto Droid. The approach under comparison implements the Q-Learning algorithm for automatically generating test cases in Android applications. However, the approach ignores the potential ability of each action, which is the essence of our approach. Four subject applications were selected in the experiment. The percentages of code coverage of the subject applications for each approach were collected and used to evaluate the effectiveness of the approaches under comparison.

Subject Applications

The selection of subject applications is based on two considerations, which are, "Are the selected subject applications representative of the type of applications for each tool?" and "Are they developed by an independent source?" The first consideration is to ensure that the subject applications are taken from a domain that represents the intention of each tool. The second consideration is to avoid bias by an interested party. An independent source of applications is the open-source community. Hence, in this experiment, four subject applications from different categories were selected from the literature based on the above considerations.

The selected subject applications are Tomdroid, Loaned, SimpleDo and Moneybalance. Tomdroid is a note-taking application. Loaned is an inventory app to keep track of personal items. SimpleDo is a to-do list application. Moneybalance tracks expenses shared by groups of people. Table 6 shows the characteristics of the subject applications comprising the number of lines, methods, classes and bytecode blocks in each application.

Experimental Setup

We implemented both approaches in the same tool, Crash Droid, to minimise the effect of different tool implementations on the results of the experiment. Crash Droid takes instrumented APK files as input to test subject applications and generates code coverage reports. The code coverage reports are generated using the JaCoCo plugin. The experiment runs Android 7.0 x 86 emulators on Windows 10 20H2 with 8 GB RAM. Table 7 shows the configuration parameters used to execute Crash Droid.

We used a similar experimental setup as described in Adamo *et al.* (2018), where we ran each approach on each subject application for two hours (i.e., 120 min). The algorithms were run ten times on each subject application to minimise the impact of randomness.

Table 4: Initial Q-values of actions

Action	Initial Q-value
a0	1.3125
a1	1.2188
b0	1.3438
b1	1.375
b2	1.2813
c0	1.2188
c1	1.1875

Table 5: Examples of rewards and Q-values for six episodes

	Episode 1		Episode 2		Episode 3		Episode 4		Episode 5		Episode 6	
Action	Reward	Q-value	Reward	Q-value	Reward	Q-value	Reward	Q-value	Reward	Q-value	Reward	Q-value
a0	3	3.687	1.5	2.1719	1	1.6407	0.75	0.75	0.75	0.75	0.75	0.75
a1	-	1.2188	-	1.2188		1.2188		1.2188	2	2.6094	1	1.5938
b0	-	1.3438	0	0	0	0	0	0	0	0	0	0
b1	0	0	0	0	0	0	0	0	0	0	0	0
b2	-	1.2813	-	1.2813	0	0	0	0	0	0	0	0
c0	-	1.2188	-	1.2188		1.2188		1.2188		0	0	0
c1	-	1.1875	-	1.1875		1.1875		1.1875		1.1875	0	0

Table 6: Characteristics of Subject Application

Application name	# Lines	# Methods	# Classes	# Blocks
Loaned v1.0.2	2837	258	70	9781
Moneybalance v1.0	1460	163	37	4959
Tomdroid v0.7.2	5736	496	131	22169
SimpleDo v1.2.0	1259	88	31	5355

Table 7: Test generation parameters

Parameter	Crash Droid	Auto Droid
Running time	120 min	120 min
Initial Q-value	1 + total weight	500

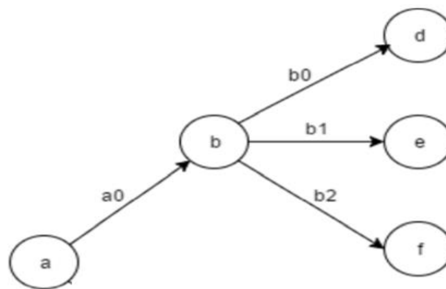

Fig. 6: An AUT represented in terms of states and actions

```
void a0(){
    ConnectivityManager connManager = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
    setTitle("State b");
    LocationListener locationListener = new MyLocationListener();
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 5000, 10, locationListener);
    locationManager.getCurrentLocation();
    int day = 4;
    switch (day) {
        case 1:
            System.out.println("Monday");
            break;
        case 2:
            System.out.println("Tuesday");
            break;
        case 3:
            System.out.println("Wednesday");
            break;
    }
}
```

RFC	CC	Net	GPS
• getSystemService • requestLocationUpdates • getCurrentLocation • setTitle • println	• Case 1 • Case 2 • Case 3	• getSystemService	• requestLocationUpdates • getCurrentLocation

```
void b0(){
    ConnectivityManager connManager = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);

    if(connManager.getAllNetworks())
    {
        setTitle("State b");
    }
}
```

RFC	CC	Net	GPS
• getSystemService • getAllNetworks • setTitle	• If	• getSystemService • getAllNetworks	

Fig. 7: Source codes for a_0 and b_0

DOI: 10.3844/jcssp.2022.90.102

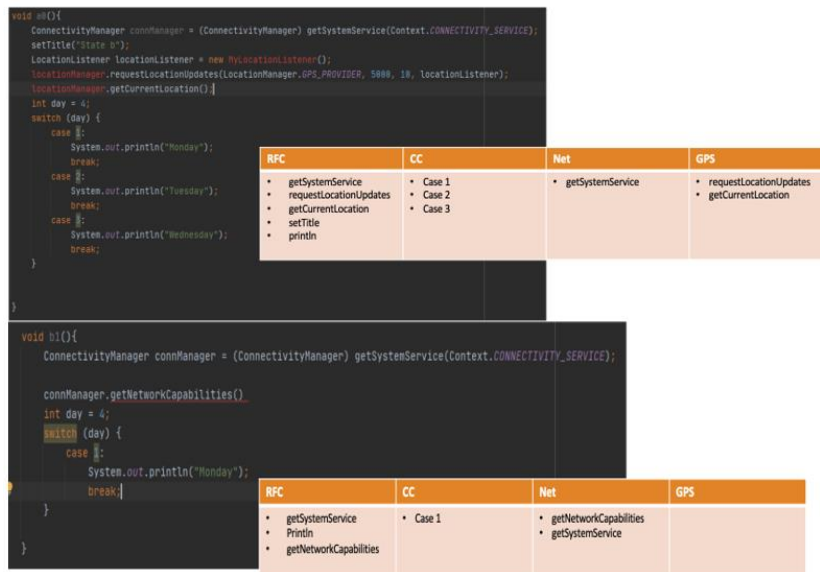

Fig. 8: Source code for a_0 and b_1

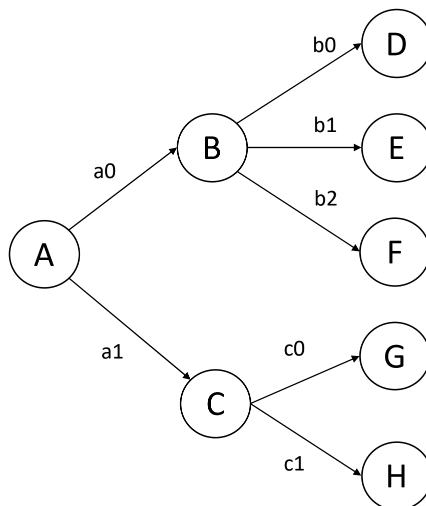

Fig. 9: Example application in terms of states and actions

Result and Discussion

Figure 10 shows a box plot of the code coverage achieved by each approach for all the subject applications on all runs. Crash Droid has a higher median code coverage compared to Auto Droid for every subject application. Crash Droid also consistently achieved a higher code coverage than Auto Droid for every subject application. The Mann-Whitney U test was performed to statistically demonstrate the significant difference between the distribution of the code coverage percentages achieved by Crash Droid and Auto Droid. The Mann-Whitney U test was chosen as it is a non-parametric statistical hypothesis test that can be used for any population distribution with two samples that are not related or that are independent. Furthermore, it is well-known that non-parametric tests are most appropriate when sample sizes are small (i.e., <100). To apply the Mann-Whitney U test, the null hypothesis is formulated as follows:

H_0: There is no significant difference between Crash Droid and Auto Droid in terms of code coverage
H_A: The code coverage of Crash Droid is greater than the code coverage of Auto Droid

The level of significance for the hypothesis tests was set to $\alpha = 0.05$ for a 2-tailed test. The Mann = Whitney U test result indicates that the differences in code coverage

of both techniques were statistically significant (U = 1223.5, p = 0.0000000273) at the $p<0.05$ significance level. Also, Crash Droid has a higher median code coverage compared to Auto Droid as shown in Fig. 11. Thus, the result suggests rejecting H_0 in favour of H_A. Based on the statistical test done, it can be concluded that at the 0.05 significance level, Crash Droid is more effective than the approach under comparison.

Threats to Validity

This section discusses the threats that can compromise the validity of an experimental study. They are the threats to the internal, external and conclusion validities.

Threats to internal validity are implementation effects that can bias the results. Faults in Crash Droid might cause such effects. To reduce these threats, Crash Droid was tested and manually inspected using the application that we developed for our case study.

Threats to external validity primarily involve the degree to which the subject applications are representative of true practice. Mitigation of these threats has been previously discussed in the Subject Applications section.

Finally, the threats to conclusion validity relate to the validity of the statistical tests. To reduce these threats, the measurements must be correct and statistical tests must be used correctly. In order to ensure that the measurements were correct due to the impact of randomness in both approaches, the experiment for each subject program was performed ten times. In the case of statistical tests, we have satisfied the statistical test assumptions of the Mann-Whitney U test.

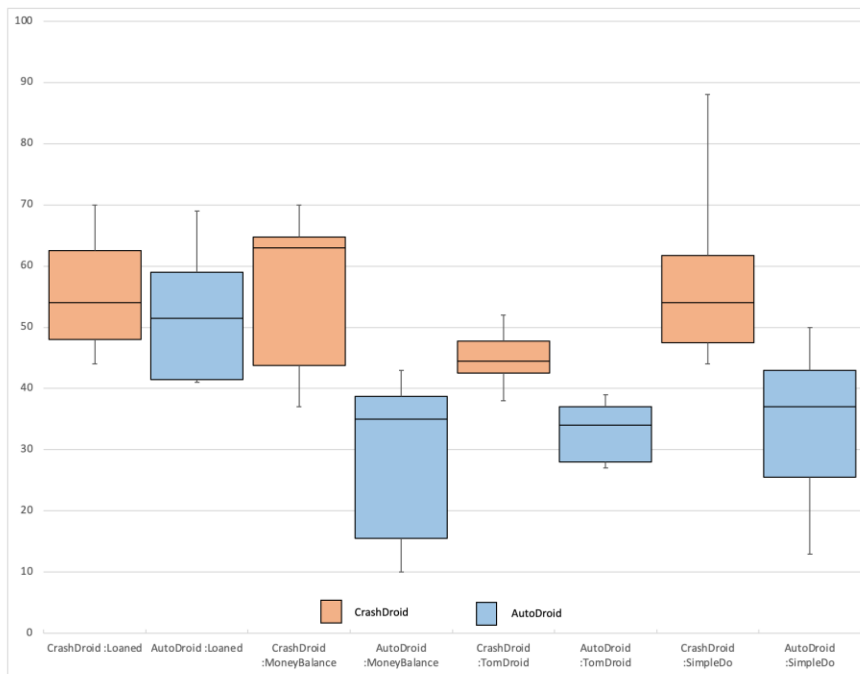

Fig. 10: Code coverage across all applications and all runs

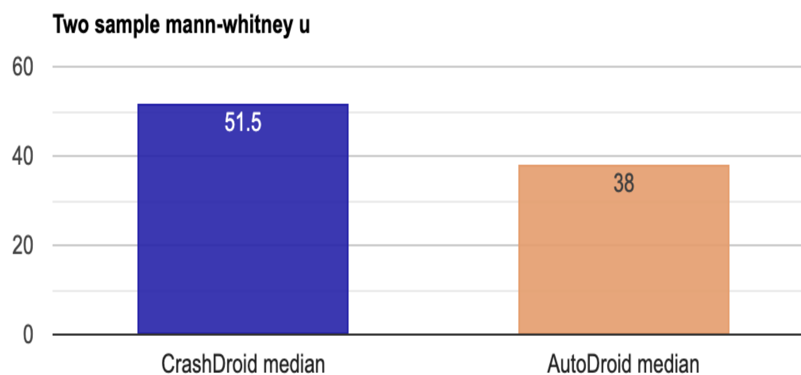

Fig. 11: Medians of the two approaches

Related Work

Research has shown that most GUI testing studies by the research community rely on the presence of a model such as the event flow graph and the finite state machine to automatically generate test cases (Banerjee *et al*., 2013). A model is crucial when aiming for test automation. However, practitioners prefer Script-based Language and the Capture and Replay Paradigm. The techniques preferred by practitioners offer limited test automation because they require manual intervention to create the test script. This difference in preference between practitioners and researchers creates a gap. The research community aims for automation, where a model is essential. However, creating a model for GUI software is time consuming and resource intensive, thereby discouraging practitioners (Memon *et al*., 2003).

Various degrees of automation are proposed by GUI testing researchers. There are techniques that are able to automate the test execution process but require human intervention to generate the test (Amalfitano *et al*., 2017). Two popular techniques in this category of automation are the Script-based and the Capture and Replay techniques. Alternatively, techniques with a higher degree of automation automates both the test generation and test execution. For the latter, test generation can be conducted offline or online. Using offline test generation, test cases must be generated before they are run. Meanwhile, with the online test generation, test cases are generated on-the-fly during the test execution. In other words, test cases are executed as they are generated. This online technique follows the strategy of *observe-select-execute*, where the possible GUI actions on an AUT's current state are observed. Using the selection strategy under consideration, one action is selected and the selected action is executed on the AUT. The advantages of this strategy are that no extraction of the GUI model is required and GUI changes in the software have no effect on the testing. Random testing has been employed in GUI testing research for many years. Fuzz testing is used to test the robustness of Windows NT applications (Forrester and Miller, 2000). Robustness is achieved if during the test, the software does not crash or hang. Fuzz testing is also applied in the UI/Application Exerciser Monkey tool. It is a command-line tool that includes the Software Development Kit (SDK) for Android ("UI/Application Exerciser Monkey," 2022). It generates pseudorandom streams of user events such as clicks, touches, gestures and a number of system-level events. (Wetzlmaier *et al*., 2016) developed a framework for random or monkey GUI testing that offers reusable components and a pre-defined, generic workflow with extension points for developing custom-built test monkeys. It supports customising SUT-specific test monkeys that randomly explore GUIs.

The random approach, however, does not explore an AUT systematically. As the actions are chosen at random, there is a high chance that actions are selected repeatedly, resulting in a lower code coverage. Also, for large GUI applications with numerous and deeply nested actions, a random algorithm is unable to test most parts of the GUI within a reasonable amount of time. To gain access to deeply nested actions and to select less-frequently executed or unexecuted actions, the probability distribution of actions over the sequence space needs to be changed. This can be achieved by using a reinforcement learning approach, in particular the Q-Learning algorithm. Q-Learning is applicable to dynamic GUI testing, as the model of the GUI is unknown until it is explored. Furthermore, the actions are generally deterministic and can be represented as a Markovian decision process.

Bauersfeld and Vos (2012; Esparcia-Alcázar *et al*., 2016) investigated the use of the Q-Learning algorithm in TESTAR (TEST Automation at the user inteRface level). TESTAR is an open-source tool that performs automated testing via the GUI itself, using the operating system's Accessibility API to recognise GUI controls and their properties and enable programmatic interaction with them. The main idea of the study was to change the probability distribution over the sequence space. Instead of a purely random selection, Q-learning selects the least frequently executed action with the purpose of exploring the GUI. The result from the investigation showed that employing the Q-learning algorithm did not significantly crash the AUT quicker but the exploration on average executes about 2.5 times as many different actions than the technique under comparison.

Mariani *et al*. (2012) proposed an automatic black-box testing tool named the Auto Black Test that automatically generates GUI test sequences. It runs on the IBM Rational Functional Tester. The tool works by exploring a GUI and assigning values to edges based on a reward function and a Q function. The reward function measures the amount of change that takes place in a GUI's state. The more the changes, the higher the reward.

Adamo *et al*. (2018) described an approach that implements the Q-Learning algorithm for automatically generating test cases in Android applications. The approach generates test cases by selecting an event with the highest Q-value among a set of available events in that particular state. Using this approach, the reward function assigns its highest reward when an event is executed for the first time.

The DroidBotx (Yasin *et al*., 2021) is another tool that was developed for generating GUI test cases based on the Q-Learning algorithm. When generating test cases, the tool selects actions from the new states with the aim of maximising the instruction, method and activity coverage by minimising any redundant execution of events. Again, the Q-function calculates the expected future rewards for actions based on the set of states it visited.

We extend and adapt the work of Bauersfeld and Vos (2012; Esparcia-Alcázar *et al*., 2016) to automate Android mobile GUI testing that includes the generation and the execution of test cases. Based on the literature, the Q-Learning algorithm was used in GUI testing and it showed better results at improving the random exploration strategy. The core purpose of using Q-Learning is to intelligently guide the action selection with the purpose of guiding the exploration of the GUI to minimise the selection of previously-selected actions and increase code coverage. However, a common limitation of previous techniques is that the highest value is assigned to the action that is executed for the first time. Besides, the selection of actions during test execution is only based on the least frequently executed action without taking into consideration its potential with respect to testing. Without this consideration, previous techniques assign a constant value to the initial action. Hence, in the initial state of the test execution, the selection of the first action is done randomly. In our study, we are proposing an approach that takes into consideration the potential of every action.

Conclusion and Future Work

We have presented the conceptual design and the result of initial experimental study of Crash Droid. Crash Droid is an automated Android GUI testing tool based on the Q-Learning algorithm. This study describes the ongoing research that aims to improve the exploration strategy of the Q-Learning algorithm by providing a mechanism to compare the ability of every action in detecting crashes. Our initial investigation shows that the ability to differentiate the potential of every action helps to achieve higher code coverage than the one that ignore it. In future, we will investigate the capability of our approach in detecting crashes. Also, we will investigate the use of other metrics for calculating the weight of every action, in particular the metrics under the context-aware metric.

Acknowledgment

The authors would like to acknowledge the Ministry of Higher Education Malaysia (MOHE) for the financial support under the Fundamental Research Grant Scheme (FRGS); Project code: FRGS/1/2019/ICT01/UPM/02/6. Also, many thanks to Dr. Tanja Vos and Dr. Johanna Ahmad for their constructive comments and suggestions, particularly in preparing the manuscript.

Author's Contributions

Goh Kwang Yi: Implemented the tool, performed the experimental study.

Salmi Binti Baharom and Jamilah Din: Contributed to the preparation of the manuscript. Participated in the experimental study and performed data analysis based on the collected data.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Adamo, D., Khan, M. K., Koppula, S., & Bryce, R. (2018). Reinforcement learning for android GUI testing. In A-TEST 2018 - Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection and Evaluation, Co-located with FSE 2018 (pp. 2–8). doi.org/10.1145/3278186.3278187

Amalfitano, D., Amatucci, N., Memon, A. M., Tramontana, P., & Rita, A. (2017). The Journal of Systems and Software A general framework for comparing automatic testing techniques of Android mobile apps. The Journal of Systems & Software, 125, 322–343. doi.org/10.1016/j.jss.2016.12.017

Banerjee, I., Nguyen, B., Garousi, V., & Memon, A. (2013). Graphical User Interface (GUI) testing: Systematic mapping and repository. Information and Software Technology, 55(10), 1679–1694. doi.org/10.1016/j.infsof.2013.03.004

Bauersfeld, S., & Vos, T. (2012). A Reinforcement Learning Approach to Automated GUI Robustness Testing. In 4th Symposium on Search Based-Software Engineering (SSBSE2012), 7–12. http://selab.fbk.eu/ssbse2012/documents/SSBSE_2012_Fast_Abstracts.pdf#page=17

Buzdalov, M., & Buzdalova, A. (2013). Adaptive selection of helper-objectives for test case generation. 2013 IEEE Congress on Evolutionary Computation, CEC 2013, 2245–2250. doi.org/10.1109/CEC.2013.6557836

Carino, S., & Andrews, J. H. (2016). Dynamically testing GUIs using ant colony optimization. Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, 138–148. doi.org/10.1109/ASE.2015.70

Esparcia-Alcázar, A. I., Almenar, F., Martínez, M., Rueda, U., & Vos, T. (2016). Q-learning strategies for action selection in the TESTAR automated testing tool. 6th International Conferenrence on Metaheuristics and Nature Inspired Computing (META 2016), 130-137.

Forrester, J. E., & Miller, B. P. (2000). An empirical study of the robustness of Windows NT applications using random testing. Proceedings of the 4th USENIX Windows System Symposium, (August), 59–68. doi.org/10.1145/1145735.1145743

Inukollu, V., Keshamoni, D., Kang, T., & Inukollu, M. (2014). Factors Influncing Quality of Mobile Apps: Role of Mobile App Development Life Cycle. International Journal of Software Engineering & Applications, 5. doi.org/10.5121/ijsea.2014.5502

Koroglu, Y., & Sen, A. (2018). QBE: QLearning-Based Exploration of Android Applications. 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 105–115. doi.org/10.1109/ICST.2018.00020

Mariani, L., Pezzè, M., Riganelli, O., & Santoro, M. (2012). Auto Black Test: Automatic black-box testing of interactive applications. Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012, 81–90. doi.org/10.1109/ICST.2012.88

Memon, A. M., Banerjee, I., & Nagarajan, A. (2003). GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. Wcre, 3, 260.

Nguyen, B. N., Robbins, B., Banerjee, I., & Memon, A. (2014). GUITAR: An innovative tool for automated testing of GUI-driven software. Automated Software Engineering, 21(1), 65–105. doi.org/10.1007/s10515-013-0128-9

Robinson, B., & Brooks, P. (2009). An initial study of customer-reported GUI defects. IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2009, 267–274. doi.org/10.1109/ICSTW.2009.22

Statista.com. (2019). Number of available applications in the Google Play Store from December 2009 to December 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

Statista.com. (2021a). Number of apps available in leading app stores as of 4th quarter 2020. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

Statista.com. (2021b). Number of mobile (cellular) subscriptions worldwide from 1993 to 2019(in millions). https://www.statista.com/statistics/262950/global-mobile-subscriptions-since-1993/

Statista.com. (2021c). Number of mobile app downloads worldwide from 2016 to 2020 (in billions). https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/

Statista.com. (2021d). Number of smartphone users worldwide from 2016 to 2021 (in billions https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

UI/Application Exerciser Monkey. (2022.). https://developer.android.com/studio/test/monkey.html

Wetzlmaier, T., Ramler, R., & Putschogl, W. (2016). A Framework for Monkey GUI Testing. Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, 416–423. doi.org/10.1109/ICST.2016.51

Yasin, H. N., Hamid, S. H., & Raja Yusof, R. J. (2021). DroidbotX: Test Case Generation Tool for Android Applications Using Q-Learning. Symmetry. doi.org/10.3390/sym13020310

Yue, S., Yue, S., & Smith, R. (2016). A Survey of Testing Context-aware Software: Challenges and Resolution.