SEMARAK ILMU PUBLISHING

Semarak Engineering Journal

Journal homepage:
https://semarakilmu.my/index.php/sej/index
ISSN: 3036-0145

Comparative Study on Code Complexity Metric to Guide Action Selection of Automated GUI Testing based on Q-Learning Algorithm

Nor Hayati Abd Razak[1], Salmi Baharom[1,*], Goh Kwang Yi[1]

[1] Department of Software Engineering and Information Systems, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Malaysia

ARTICLE INFO

Article history:
Received 20 November 2024
Received in revised form 29 December 2024
Accepted 10 February 2025
Available online 20 March 2025

Keywords:
Android; automated GUI Testing; code complexity metrics; Q-learning

ABSTRACT

Poor-quality mobile applications will disappoint users, resulting in their uninstallation of the applications. Due to mobile applications' event-driven and gesture-based nature, several researchers have proposed GUI testing as an alternative to system testing. Furthermore, since GUI testing calls for mimicking user actions on the application, automation is essential for replicating how humans interact with the GUI widgets. The *observe-select-exec*ute strategy is used in automated GUI testing tools to observe all GUI actions in their current state, select one, and execute it. In the *observe-select-execute* strategy, few researchers have proposed a Q-Learning algorithm to guide the exploration of GUIs. However, the exploration solely looks at the least frequent action without considering the action's potential ability to detect failures. We propose a method that compares actions by their weight to improve GUIs exploration, leading to effectively detecting crashes and achieving a higher code coverage. This paper describes a comparative study of code complexity metrics to identify the potential action. The results statistically demonstrate that the code complexity metric can improve the exploration of GUIs based on the percentage of AUT code coverage collected. The combination of RFC and CYC metrics outperforms the other four metrics under comparison.

1. Introduction

The growing demand for Android applications motivates developers to develop a wide range of applications that cater to customer needs [1]. Considering that billions of applications were developed and downloaded globally, testing should be necessary to ensure the application's quality. The competition among developers to frequently release their applications makes testing a lesser priority or ignored entirely to expedite the development process, making it more difficult to guarantee the quality of applications. Furthermore, testing Android applications is costly, time-consuming, and challenging. Thus, researchers have suggested automation as one of the solutions to accelerate the process of testing [2].

* *Corresponding author.*
E-mail address: salmi@upm.edu.my

https://doi.org/10.37934/sej.8.1.2336b

Mobile applications depend highly on their Graphical User Interfaces (GUI) due to their event-driven nature and gesture-based interactions. As a result, GUI testing commonly replaces system testing in mobile application testing. GUIs involve selecting an event that exercises a GUI widget (i.e., test case), executing the selected event (i.e., test execution), and monitoring resulting changes to the software state (i.e., test oracle). In other words, GUI testing necessitates simulating user actions on applications, and automated GUI testing mimics human interaction with GUI widgets. One strategy in automated GUI testing tools is to use the observe-select-execute approach. It begins with the launch of the Application Under Test (AUT). It then observes the GUI actions on the AUT's current state, selects an action from the observed GUI actions, and executes the selected action. The strategy's primary goal is to choose an action that could lead to new and desired GUI states.

Studies by Adamo *et al.,* [3] and Yassin *et al.,* [4] highlighted that random testing is commonly used in automated approaches for Android GUI testing. The main advantage of random testing is that no prior knowledge is required. However, random testing does not ensure that the application under test is systematically explored. It is more likely that previously chosen actions will be chosen again, resulting in lower code coverage and unrevealed failures. To overcome the limitation of the random technique, studies by Pan *et al.,* [5], Adamo *et al.,* [3], and Voung *et al.,* [6] have proposed a Q-Learning algorithm to guide the GUI exploration systematically. However, the approach solely favors the least frequent action and does not consider the potential ability of action to disclose failures.

Code complexity has been employed by Zhuang [7], Gao *et al.,* [8], and Gezici *et al.,* [9] to measure the quality of mobile applications. Code complexity metrics have been widely used to predict the number of defects in the code, as complex code is more prone to bugs. Thus, employing code complexity to guide the selection of potential action in exploring Android applications can be beneficial to effectively revealing defects in automated GUI testing, particularly for those who employ the observe-select-execute strategy.

This paper describes a comparative study of code complexity metrics for calculating an action's weight to guide the GUI exploration of Android applications. This research aims to develop a testing tool that can automatically detect crashes in Android Applications. We named the tool Crash Droid [24]. It is based on the observe-select-execute strategy, which employs the Q-Learning algorithm to compare actions based on their potential abilities to uncover failures.

1.1 Related Work

Statista [11] has reported that 71.8% of mobile users prefer Android operating systems worldwide. The demand for Android applications drives developers to create more applications as quickly as possible. Nevertheless, testing's importance should be noted for quality assurance. Studies reported that many applications suffer poor user experience caused by insufficient testing before release [8,12]. The poor user experience would frustrate users. According to ww.buildfire.com, 71% of users stopped using applications within 90 days of downloading them due to dissatisfaction with the applications. Thus, Android application testing is critical to ensuring quality and user satisfaction. Since most interactions with a mobile app happen through its graphical user interface (GUI), GUI testing is essential to make sure users from becoming dissatisfied [13]. A well-designed GUI can ensure that the interaction between the user and the software is as smooth and easy as possible [14,15]. GUI testing ensures that functionality and business specifications are met and that no crashes occur when users use it.

GUI testing can be done automated or manually. A tester performs user operations on the target application and verifies correctly during manual testing. Manual testing is time-consuming, tedious, and prone to error. Additionally, manual testing does not allow 100% coverage and in-depth

execution. Conversely, in an automated approach, all test cases are executed automatically. Due to the time and effort required to develop GUI test cases in Android applications, which require a highly competitive environment and repeated testing on a variety of different devices, [14,15,12] and [13] have implemented their study using Automated GUI Testing. The most important feature of automated testing is to improve code coverage to detect crashes [6,16].

With automated testing, no human interaction is required during the process, and it could significantly reduce maintenance costs and testing time [12,17,18]. Previous researchers for GUI testing have introduced several approaches. Model-based approaches to testing Android apps are popular because they generate test cases based on a model that reflects the AUT's behavior ([12]). This approach can be made manually or automatically, but most researchers prefer to do it automatically because it saves time. [2] developed a tool named CrawlDroid to improve coverage of GUI Testing by widgets together based on their position in a state and giving each action that the group supports a priority value. Stoat, presented by [1], utilizes an application's behavior models to refine test generation iteratively in the direction of high coverage and diverse event sequences.

While model-based testing has been used for Android GUI testing and has been shown to enhance code coverage and error detection, one of the approach's shortcomings is that it spends a significant amount of time exploring AUTs. Besides, it has shortcomings such as poor models and limited scalability; therefore, it does not provide considerable advantages over random strategies [5]. Due to the complexity of the GUI, random testing is a common approach for Android GUI testing [3, 4]. The monkey tool, an Android framework's built-in random testing tool usually used in Random Testing. It can run a specified script at random to generate user or system events. Random testing does not require prior knowledge about the AUTs and is easy to set up. It generates random actions to investigate application behaviors and detect failures. This strategy, however, has the issue of generating random inputs to explore the potential action, which may result in meaningless actions and do not add to the potential of detecting failure. The activities, especially when triggered by system events using third-party app access, are impossible to explore during Monkey runs because Monkey always starts from the AUT rather than third-party apps. Moreover, there is no support for generating system events in the Monkey tool. According to [19], their study of Random Testing in WeChat found that 1.6% (7 out of 439) of unexplored activities are unexplored due to the requirement for inter-component communication (ICYC) between WeChat and another application.

Compared to Random Testing, Q-learning-based exploration increases code coverage and fault detection. [6,3,20,21], and [5] used a Reinforcement Learning approach called Q- learning in their study to benefit from model-based and random testing. The Q-learning approach has five components: agent, environment, state, action, and reward. The agent acts independently in the environment to attain a goal. The agent interacts with the environment and learns by trial and error. The environment is a situation in which an agent is present or surrounded, while the state describes the current status of the environment. Meanwhile, the reward is used to evaluate actions in a particular state. In summary, the agent acts during each interaction, assigns rewards, and transits to a new state.

The Q-learning-based method generates 10.30% higher coverage for the same actions than the random test execution [3]. [4] implemented DroidBotX, adopting a Q-learning approach using an Upper Confidence Bound (UCB) to improve coverage and crash detection. [20] introduce QBE, a fully automated black-box testing methodology that uses Q-Learning to explore GUI actions. They collect coverage with ELLA, a binary instrumentation tool.

Although many researchers employed Q-learning in their studies, it still has some limitations. The limitation of Q-learning is that the exploration is only based on the least frequent action and does not explore how each action can identify potential value in disclosing the failures. The learning

process will take longer if it involves an extensive system. Hence, [10] proposed an approach that guides the GUIs exploration by considering every action's potential.

1.2 Code Complexity Metrics

It is commonly acknowledged that the code's complexity influences the software developer's mental burden, which causes a greater number of human errors and subsequently introduces more faults to the code. Hence, many researchers have investigated the ability of code complexity metrics to predict the number of faults revealed during software testing and operation. Various code complexity metrics have been suggested to measure the complexity of mobile applications, such as Line of Codes (LOC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Response for Class (RFC), Halstead Metrics (HM), Weighted Method for Class (WMC) and McCabe Cyclomatic Complexity (CYC). Each of these metrics has its place. For this paper, we have selected three popular metrics used in the previous studies of mobile applications, as shown in Table 1.

Table 1
Code complexity metrics used in mobile applications

Researcher	DIT	NOC	LCOM	CBO	RFC	WMC	LOC	CYC
[9]	/	/	/	/	/	/		
[8]	/	/			/			
[25]							/	
[28]						/		
[26]						/		/
[27]						/		/
[30]					/	/		
[29]						/		/

The three selected metrics are described as follows:

1.2.1 McCabe Cyclomatic Complexity (CYC)

CYC is one of the most widely used metrics for measuring the complexity of a code. It is a quantitative measure of paths through the source code that are linearly independent. If the program has a high complexity number, the probability of error is high, and detecting the error is hard. The Control Flow Graph is used to compute CYC (CFG). Given a CFG, the CYC metric is computed as E - N + 2P, where E represents the number of edges, N represents the number of nodes, and P represents the number of modules in the program. An alternative calculation of CYC is based on a program's number of decision points. The decision points are if, for, for-each, while, do, and case statements in a program.

1.2.2 Response for Class (RFC)

RFC is the total number of methods that can potentially be executed in response to a message received by an object of a class. This is the total number of class methods, and all distinct methods are called directly within the class. If RFC increases, class design complexity increases and becomes hard to understand due to a large number of class methods.

1.2.3 Weighted Method per Class (WMC)

WMC was introduced by [22] as the sum of the complexity of all methods declared in a class. Since the methods of the parent are inherited by the child, many methods in a class may have a potentially greater impact on the children of the class. The WMC metric can be combined with other metrics, such as CYC, to obtain information about class complexity. Consider a class C1, with method M1…Mn defined in a class. Let c1... cn be the complexity of the methods, then WMC is given as follows

$$\sum_{i=1}^{n} ci \tag{1}$$

where c_i is the complexity of the methods associated with the i^{th} class.

1.3 Q-Learning Algorithm

Q-learning is a reinforcement learning policy that attempts to explore a strategy that maximizes total reward by determining the best action selection in the current state. According to Cheung *et al.,* [23], an Android app can be designed using Markov Decision Process (MDP). MDP models decisions with probabilistic and deterministic rewards and penalties. MDP's main components are agents, states, rewards, a series of actions in every state, and an action-value function. At each stage of the process, the agent may choose an action available in the current state, resulting in the current state transitioning to the next state and offering a reward for the action.

1.3.1 Reward function

The reward function computes the reward value of an action that moves from one state to another. The function enables the agent to evaluate potential actions to identify crashes. Actions with high value have greater potential. The reward function R determines the value of the reward

$$R(a, s, s') = \begin{cases} r_{init}, & \text{if } x_a = 0 \\ \frac{1}{x_a} \times a_{s'}, & \text{otherwise} \end{cases} \tag{2}$$

where:
r_{init} is the initial default reward.
x_a is the frequency with which the action has been carried out in the state.
$a_{s'}$ is the number of actions in state s' that were not in state s

This study employs a different default reward than other studies. The initial value is computed based on the initial value of actions plus one instead of a constant value. The goal is to speed up crash detection by guiding action selection from the first execution. Later activities are rewarded based on the least frequently chosen actions during testing. The more frequently an action is completed, the less interesting it becomes to the agent. For this study, the reward function analyses actions with high reward values and more potential, and the frequency requirement prevents selecting an action repeatedly.

1.3.2 Q-value function

The action-value function is used to compute the value of an action, that exists in a specific state of an AUT. It considers the value of the current reward earned for executing the action and the best future value linked with the action. This function is critical since it allows the tool to plan ahead of time when deciding which action to take in a certain condition, which will favor application exploration. The q-value function is defined as follows

$$Q(s,a) = R(s,a,s') + \gamma \cdot max_{a^* \in A_{s'}} Q(s', a^*) \quad (3)$$

where:

Q(s, a)	=	action's value of an in-state s
R(s, a ,s')	=	value of reward for performing action a in state s
$max_{a* \in As'}$Q(s 0 , a∗)	=	action a's maximum value in the state when action a is executed
γ	=	discount factor

The discount factor determines the effect of future rewards in calculating the action-value function for 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.

2. Methodology

2.1 The Tool: Crash Droid

We used Crash Droid [11] as the tool for the comparative study. Figure 1 depicts the tool's overview.

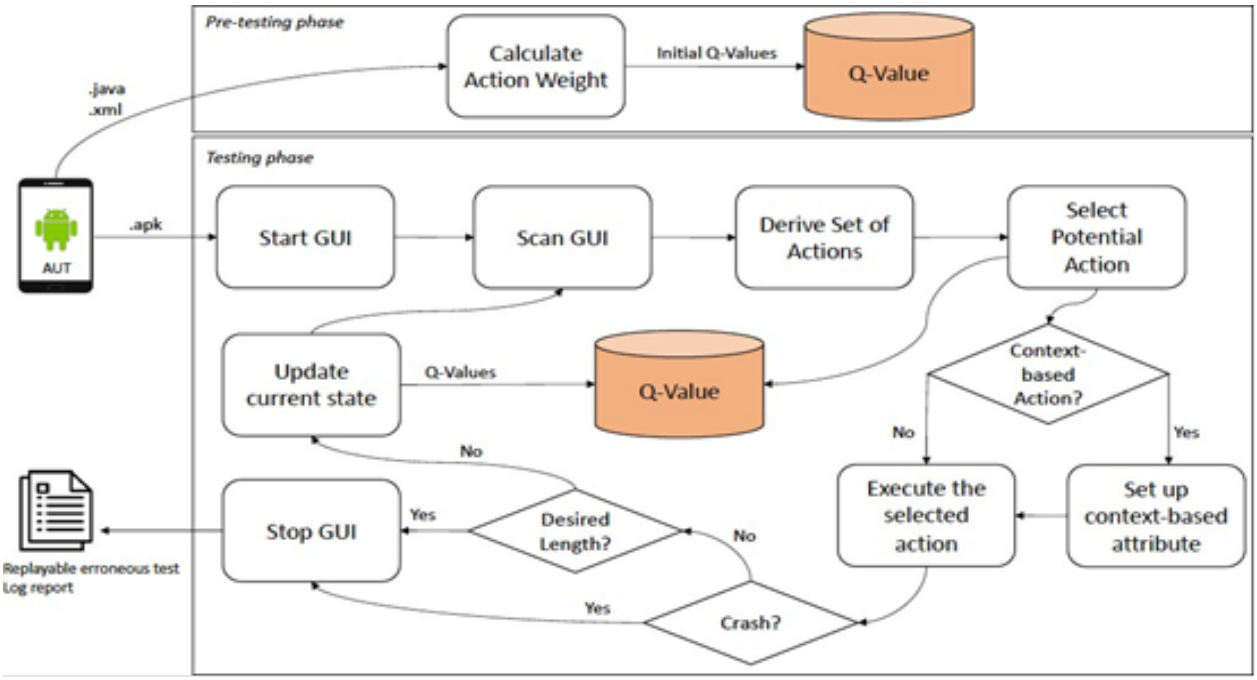

Fig. 1. Overview diagram of Crash Droid

The observe-select-execute strategy is used by the tool when interacting with an AUT. All possible GUI actions on the current state of the AUT are observed, one action is chosen based on its potential for crash detection, and the chosen action is then executed on the AUT. In order to maximize coverage and locate crashes, the application further explores the GUI using the Q-Learning algorithm. Crash Droid operates on the Windows platform. It is created using numerous programming languages, such as Java, Batch script, and Visual Basic.Net (VB.Net). There are two phases to the tool. The first is called pre-testing, where Crash Droid calculates each action taken from the user interface file beforehand. The tool adopts the Q-Learning and the Jaccard Distance algorithms in the second step to test the AUT.

2.1.1 Case study

The idea of the Q-Learning algorithm is illustrated using an example of an Android application. The Android application is represented in the perspective of states and actions. Figure 2 depicts the Android application's eight states (a to g) and seven actions (a0, a1, b0, b1, b2, c0, c1). a0 and a1 are two possible actions for state a, whereas states d, e, f, g, and h have no possible actions that result in actions to end the state. The direction of the arrow shows how execution moves from the current state to the new state. Crash Droid determines action weights as the initial value instead of using a constant number. The tool uses the code complexity metric to calculate the action weight. The action weight is then added by 1 to determine the initial Q-value. Assume that Table 2 shows the initial Q-value of AUT depicted in Figure 2.

We illustrate the Crash Droid testing phase using the information from Figure 2 and Table 2. The testing phase starts with episode 0, where the agent is in state a. The available actions for state a are a0 and a1. The agent runs a0 because it has a higher Q-value. State a now transit to state b. Since b has not yet been terminated, the reward function is used to obtain the reward for a0, and a new Q-value for a0 is obtained using the Q-value function. Between three actions: b0, b1, and b2, in state b, the agent executes b1 since it has the highest Q-value. Now, state b transitions to state e. The Q-value and reward value for b1 are now 0 because state e has terminated. The agent continues this process until the termination action that closes the AUT is carried out. The reward and Q-value for each action after each episode are detailed in Table 3.

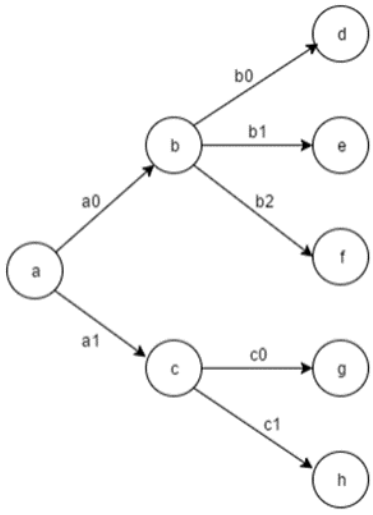

Fig. 2. Example of AUT in perspective of states and actions

Table 2
Actions initial value

Actions	Initial Q-Value
a0	2.00
a1	1.12
b0	1.35
b1	1.35
b2	1.15
c0	1.44
c1	1.38

Table 3
Rewards and Q-Values on each episode

	Episode 0		Episode 1		Episode 2		Episode 3	
Action	Reward	Q-value	Reward	Q-Value	Reward	Q-Value	Reward	Q-Value
a0	3	3.40	1.50	2.18	1.00	1.44	0.75	0.75
a1	-	1.12	-	1.12	-	1.12	2.00	2.22
b0	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
b1	-	1.35	0.00	0.00	0.00	0.00	0.00	0.00
b2	-	1.15	-	1.15	0.00	0.00	0.00	0.00
c0	-	1.44	-	1.44	-	1.44	0.00	0.00
c1	-	1.38	-	1.38	-	1.38	-	1.38

2.2 The Action Weight Calculation

For this experiment, we use the metrics discussed in the previous section (i.e., RFC, CYC, WMC), together with two calculated code metrics. The two calculated code metrics are derived from a combination of two metrics which are RFC and CYC; and WMC and CYC.

The action weight calculation based on the five metrics is explained as follows:

2.2.1 The action weight using RFC Metric

The RFC metric counts the number of different methods calls in each action in AUT. If a method is invoked many times, only the first time is counted. Using the RFC metric, the weight for action, WRFCA, is calculated by dividing the number of methods called by the code in each action, NRFC, by the greatest number of methods called by each action in the AUT, MRFC. WRFCA is calculated as follows

$$W_{RFCA} = N_{RFC} \div M_{RFC} \quad (4)$$

where
N_{RFC} = RFC complexity of the code in each action
M_{RFC} = the largest RFC complexity of the code among actions in the AUT

2.2.2 The action weight using CYC Metric

The Cyclomatic Complexity (CYC) metric measures the number of linearly independent pathways a code can traverse. With the CYC metric, the weight of action is determined by dividing the

complexity of the code in each action, NCYC, by the largest complexity of actions in the AUT, MCYC. WCYCA is calculated as follows

$W_{CYCA} = N_{CYC} \div M_{CYC}$

where
N_{CYC} = CYC complexity of the code in each action
M_{CYC} = the largest CYC complexity of the code among actions in the AUT

2.2.3 The action weight using WMC Metric

The WMC metric calculates all methods in each action. The weight for each action based on the WMC metric, WWMCA, is found by dividing the number of methods called by the code in each action, NWMC, by the total number of methods called by all actions in the AUT, MWMC. WWMCA is calculated as follows

$W_{WMCA} = N_{WMC} \div M_{WMC}$ (6)

where
N_{WMC} = WMC complexity of the code in each action
M_{WMC} = the largest WMC complexity of the code among actions in the AUT

2.2.4 The Calculated Metrics

For the calculated metrics, we sum the two metrics' weights metric. The weight for a combination of WMC and CYC is as follows

$W_{WMCYC} = W_{WMCA} + W_{CYCA}$ (7)

where
W_{WMCA} = The weight of WMC
W_{CYCA} = The weight of CYC

The weight for the combination of RFC and CYC metrics is as follows
$W_{RFCYCA} = W_{RFCA} + W_{CYCA}$ (8)

where
W_{RFCA} = The weight of RFC
W_{CYCA} = The weight of CYC

3. Result

The significance of the code complexity metric has been investigated to improve the GUI's exploration of the Q-Learning algorithm by comparing the code coverage percentages achieved by the compared metrics. The experiment used five code complexity metrics and five Android applications. The code coverage percentage of subject applications for each code complexity metric

was collected and analyzed to determine the most appropriate code complexity metric to identify potential actions for detecting crashes in Android applications for automated GUI testing.

3.1 Subject Applications

The applications for each subject are chosen based on two considerations: "Are the selected subject applications representative of the type of applications for each tool?" and "Does an independent source develop them?" The first consideration is to ensure that the subject applications are drawn from a domain representing each tool's intention. Since the second consideration is to avoid bias from vested interest, the applications are selected from the open-source community. Five subject applications from various categories were chosen for this experiment based on the above considerations.

The Tomdroid, Droidshows, SimpleDo, Loaned, and Moneybalance were chosen to be the subject applications of this experiment. The app Loaned helps users monitor their belongings, while SimpleDo organizes tasks application. Tomdroid is an application for taking notes, while Droidshows is a TV series browser and tracker, and Moneybalance keeps track of group-shared spending. The subject apps' characteristics such as name, version, code blocks, methods, and classes are shown in Table 4. Apps range from 4,959 to 22,169 code blocks, with 116 to 744 methods and 31 to 168 classes. We obtain code coverage information from the coverage report produced by the Jacoco plugin.

Table 4
Applications used in the experiment and their details

Application Name	Version	# Lines	# Methods	# Classes	# Blocks
Tomdroid	0.72	5011	744	168	22169
Droidshows	6.5	3714	516	89	16224
Simpledo	1.2.0	943	116	31	5355
Loaned	1.0.2	2034	344	73	9781
Moneybalance	1	1677	300	55	4959

3.2 Experimental Setup

The experiment is carried out using emulators for Android Pixel 2 Pie 9.0 - API 28 running on Windows 10 20H2 with 8 GB of RAM. We tested each code complexity metric for every subject application for 120 minutes (2 hours). In order to minimize the impact that randomness had on the results, the experiment was repeated ten times on each subject application for each metric of code complexity.

3.3 Code Coverage

Code coverage is used to determine the exploration of AUT. High code coverage shows that the metric can cover a significant portion of the AUT. The effectiveness of Q-learning is studied using various code complexity metrics. The code complexity metrics value for each action in states influences the decision of which action to execute. Thus, the experiment was conducted to obtain the coverage results regarding the class, lines, and branch coverages.

3.4 Results and Discussion

Table 5 shows the average code coverage of class, line, and branch achieved by each AUT during execution using three bases and two calculated code complexity metrics.

Table 5
Average code coverage among subject application

Application Name	RFC	CYC	WMC	RFC + CYC	WMC + CYC
Moneybalance	15.33	15.33	15.33	15.33	15.33
Tomdroid	48	50	50	50	39.33
Droidshow	25.67	31.67	23.33	34.67	21.33
Loaned	10.67	8.67	38.67	24.67	34.67
Simpledo	18	23.67	25.67	27.67	18

Figure 3 shows a box plot of complexity metrics for all AUT running the experiment. RFC + CYC has a higher median code coverage than other code complexity metrics.

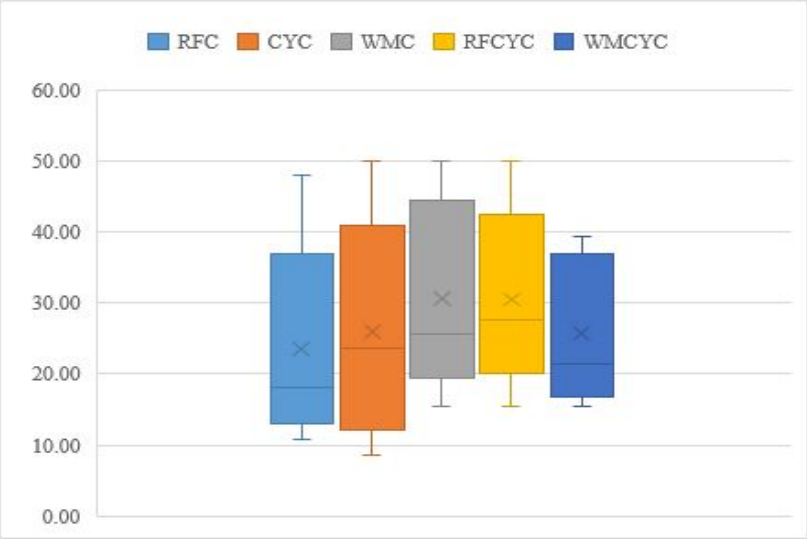

Fig. 3. Code complexity metric across all applications

The significant distribution of code complexity metrics in AUT code coverage was statistically demonstrated using the Kruskal-Wallis H test. The Kruskal-Wallis H test was selected because it is a non-parametric test that permits comparisons between more than two populations in a completely randomized design. In addition, applying the non-parametric test is appropriate when the sample size of the population is too small. The null hypothesis for the Kruskal-Wallis H test is as follows:

H0: The distribution of AUTs code coverage is the same
HA: The distribution differs in code complexity metric

The significance level for right-tailed chi-square tests was set to = 0.05 for hypothesis tests. According to the results of the Kruskal-Wallis H test, there are differences in code coverage for five AUTs that are statistically significant (H = 1.9938, p = 0.2971, df = 4). The result indicates that H0 should be rejected in favor of HA. At the 0.05 level of significance, the statistical test indicates that there are differences in code coverage between the five AUTs. Based on the experimental results, code complexity metrics can improve the exploration of GUIs based on the percentage of AUT code

coverage collected. Also, the experiment has shown that the combination of RFC and CYC metrics outperforms other code complexity metrics in terms of code coverage. Therefore, it is statistically proven that the complexity of the code behind each action is an important factor in determining its potential to discover failures.

3.5 Threats to Validity

This section discusses the potential threats to an experiment's validity. The first threat, external validity, involves the degree to which the experimental study's subject applications should represent actual practice. However, this threat's mitigation has previously been discussed in the Subject Applications section. Second, implementation effects can bias the results' internal validity. Faults in Crash Droid might cause such effects. Thus, Crash Droid was tested and manually inspected to reduce the threat. Finally, the threats to conclusion validity relate to the validity of the statistical tests. This threat can be reduced by ensuring the correctness of the measurements and the use of statistical tests must be correct. To ensure the correctness of the measurement due to the impact of randomness in both approaches, the experiment for every subject program was performed ten times. For the statistical tests, we ensure the assumptions of the Kruskal-Wallis H test have been satisfied.

4. Conclusions

Mobile application testing is becoming increasingly important as more and more people rely on their mobile devices to get their work done. As mobile users interact with the application on numerous operating systems and devices, especially Android, there are additional aspects to consider when testing mobile applications. Thus, the developer often puts mobile testing aside due to the time and effort it takes. Q-learning dynamically automates GUI test generation for Android applications, saving humans time and effort in analyzing and writing test cases. Although the Q-learning approach leverages the concept of dynamic action selection to intelligently choose an action from the available actions, Q-learning exploration is only based on the least frequent action. It does not explore how each action can identify potential value in disclosing the failures. Therefore, in this study, the Q-learning algorithm is enhanced by assigning the value for an action executed for the first time rather than selecting the initial action randomly. We evaluated the code coverage acquired by the proposed approach to assessing the difference in actions' potential abilities during Android application testing. The findings demonstrate a difference in AUT code coverage among other code complexity metrics. In the future, the effectiveness of the proposed approach will be evaluated with other state-of-the-art tools to investigate the capability of detecting crashes. In addition, the initial value of each potential action should be considered, not only the initial value for the first action execution.

Acknowledgement

The authors would like to thank Assoc. Prof. Dr. Koh Tieng Wei for your insightful comments and suggestions especially regarding the preparation of the experiment results. This work was supported by the Fundamental Research Grant Scheme (FRGS/1/2019/ICT01/UPM/02/6, 2019).

References

[1] Su, Ting, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. "Guided, stochastic model-based GUI testing of Android apps." In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 245-256. 2017. https://doi.org/10.1145/3106237.3106298

[2] Cao, Yuzhong, Guoquan Wu, Wei Chen, and Jun Wei. "Crawldroid: Effective model-based gui testing of android apps." In *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, pp. 1-6. 2018. https://doi.org/10.1145/3275219.3275238

[3] Adamo, David, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. "Reinforcement learning for android gui testing." In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, pp. 2-8. 2018. https://doi.org/10.1145/3278186.3278187

[4] Yasin, Husam N., Siti Hafizah Ab Hamid, and Raja Jamilah Raja Yusof. "Droidbotx: Test case generation tool for android applications using Q-learning." *Symmetry* 13, no. 2 (2021): 310. https://doi.org/10.3390/sym13020310

[5] Pan, Minxue, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. "Reinforcement learning based curiosity-driven testing of android applications." In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pp. 153-164. 2020. https://doi.org/10.1145/3395363.3397354

[6] Vuong, Thi Anh Tuyet, and Shingo Takada. "A reinforcement learning based approach to automated testing of android applications." In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pp. 31-37. 2018. https://doi.org/10.1145/3278186.3278191

[7] Zhuang, Yan. "The performance cost of software obfuscation for android applications." *Computers & Security* 73 (2018): 57-72. https://doi.org/10.1016/j.cose.2017.10.004

[8] Gezici, Bahar, Ayça Tarhan, and Oumout Chouseinoglou. "Mobil uygulamaların evriminde karmaşıklık, boyut ve iç kalite gelişimi: Keşifsel bir çalışma." *Gazi Üniversitesi Mühendislik Mimarlık Fakültesi Dergisi* 34, no. 3 (2018): 1483-1500. https://doi.org/10.17341/gazimmfd.460547

[9] Gao, Jun, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. "On the evolution of mobile app complexity." In *2019 24th international conference on engineering of complex computer systems (ICECCS)*, pp. 200-209. IEEE, 2019. https://doi.org/10.1109/ICECCS.2019.00029

[10] Yi, Goh Kwang, Salmi Binti Baharom, and Jamilah Din. "Improving the Exploration Strategy of an Automated Android GUI Testing Tool based on the Q-Learning Algorithm by Selecting Potential Actions." *Journal of Computer Science* 18, no. 2 (2022): 90-102. https://doi.org/10.3844/jcssp.2022.90.102

[11] Statista. "Global Mobile OS Market Share 2022 | Statista," February 21, 2023.https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/#statisticContainer

[12] Kong, Pingfan, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. "Automated testing of android apps: A systematic literature review." *IEEE Transactions on Reliability* 68, no. 1 (2018): 45-66. https://doi.org/10.1109/TR.2018.2865733

[13] Di Martino, Sergio, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. "Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing." *Software Testing, Verification and Reliability* 31, no. 3 (2021): e1754. https://doi.org/10.1002/stvr.1754

[14] Coppola, Riccardo. "Fragility and evolution of android test suites." In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 405-408. IEEE, 2017. https://doi.org/10.1109/ICSE-C.2017.22

[15] Coppola, Riccardo, Maurizio Morisio, and Marco Torchiano. "Mobile GUI testing fragility: a study on open-source android applications." *IEEE Transactions on Reliability* 68, no. 1 (2018): 67-90. https://doi.org/10.1109/TR.2018.2869227

[16] Pilgun, Aleksandr, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. "Fine-grained code coverage measurement in automated black-box android testing." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, no. 4 (2020): 1-35. https://doi.org/10.1145/3395042

[17] Amalfitano, Domenico, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. "PT," 2018.

[18] Saharan, Divya, Yogesh Kumar, and Rahul Rishi. "Analytical study and implementation of web performance testing tools." In *2018 International Conference on Recent Innovations in Electrical, Electronics & Communication Engineering (ICRIEECE)*, pp. 2370-2377. IEEE, 2018. https://doi.org/10.1109/ICRIEECE44171.2018.9008408

[19] Zheng, Haibing, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. "Automated test input generation for android: Towards getting there in an industrial case." In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 253-262. IEEE, 2017. https://doi.org/10.1109/ICSE-SEIP.2017.32

[20] Koroglu, Yavuz, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. "Qbe: Qlearning-based exploration of android applications." In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 105-115. IEEE, 2018. https://doi.org/10.1109/ICST.2018.00020

[21] Vuong, Thi Anh Tuyet, and Shingo Takada. "Semantic Analysis for Deep Q-Network in Android GUI Testing." In *SEKE*, pp. 123-170. 2019. https://doi.org/10.18293/SEKE2019-080

[22] Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." *IEEE Transactions on software engineering* 20, no. 6 (1994): 476-493.https://doi.org/10.1109/32.295895

[23] Cheung, Tang Lung, Kari Okamoto, Frank Maker III, Xin Liu, and Venkatesh Akella. "Markov decision process (MDP) framework for optimizing software on mobile phones." In *Proceedings of the seventh ACM international conference on Embedded software*, pp. 11-20. 2009. https://doi.org/10.1145/1629335.1629338

[24] Goh, Kwang Yi, Salmi Baharom, Jamilah Din, and Nurfadhlina Mohd Sharef. "The Development of an Android Automated Testing Tool: CrashDroid." In *2022 Applied Informatics International Conference (AiIC)*, pp. 165-171. IEEE, 2022. https://doi.org/10.1109/AiIC54368.2022.9914593

[25] Coppola, Riccardo, Maurizio Morisio, and Marco Torchiano. "Evolution and fragilities in scripted gui testing of android applications." In *Proceedings of the 3rd International Workshop on User Interface Test Automation. ACM*. 2017. https://doi.org/10.1145/3127005.3127008

[26] Fasano, Fausto, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. "Investigating Mobile Applications Quality in Official and Third-party Marketplaces." In *ENASE*, pp. 169-178. 2019. https://doi.org/10.5220/0007757601690178

[27] Grano, Giovanni, Andrea Di Sorbo, Francesco Mercaldo, Corrado A. Visaggio, Gerardo Canfora, and Sebastiano Panichella. "Android apps and user feedback: a dataset for software evolution and quality improvement." In *Proceedings of the 2nd ACM SIGSOFT international workshop on app market analytics*, pp. 8-11. 2017. https://doi.org/10.1145/3121264.3121266

[28] Hamdi, Oumayma, Ali Ouni, Eman Abdullah AlOmar, Mel Ó. Cinnéide, and Mohamed Wiem Mkaouer. "An empirical study on the impact of refactoring on quality metrics in android applications." In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pp. 28-39. IEEE, 2021. https://doi.org/10.1109/MobileSoft52590.2021.00010

[29] Noei, Ehsan, Mark D. Syer, Ying Zou, Ahmed E. Hassan, and Iman Keivanloo. "A study of the relation of mobile device attributes with the user-perceived quality of Android apps." *Empirical Software Engineering* 22 (2017): 3088-3116. https://doi.org/10.1007/s10664-017-9507-3

[30] Pecorelli, Fabiano, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. "Testing of mobile applications in the wild: A large-scale empirical study on android apps." In *Proceedings of the 28th international conference on program comprehension*, pp. 296-307. 2020. https://doi.org/10.1145/3387904.3389256