REINFORCEMENT LEARNING-BASED TEST CASE GENERATION WITH TEST SUITE PRIORITIZATION FOR ANDROID APPLICATION TESTING

Md Khorrom Khan

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

July 2023

APPROVED:

Renee Bryce, Major Professor
Barrett Bryant, Committee Member
Stephanie Ludi, Committee Member
Hyunsook Do, Committee Member
Gergely Záruba, Chair of the Department of Computer Science and Engineering
Paul S. Krueger, Dean of the College of Engineering
Victor Prybutok, Dean of the Toulouse Graduate School

Khan, Md Khorrom. *Reinforcement Learning-Based Test Case Generation with Test Suite Prioritization for Android Application Testing*. Doctor of Philosophy (Computer Science and Engineering), July 2023, 120 pp., 24 tables, 11 figures, 131 numbered references.

This dissertation introduces a hybrid strategy for automated testing of Android applications that combines reinforcement learning and test suite prioritization. These approaches aim to improve the effectiveness of the testing process by employing reinforcement learning algorithms, namely Q-learning and SARSA (State-Action-Reward-State-Action), for automated test case generation. The studies provide compelling evidence that reinforcement learning techniques hold great potential in generating test cases that consistently achieve high code coverage; however, the generated test cases may not always be in the optimal order. In this study, novel test case prioritization methods are developed, leveraging pairwise event interactions coverage, application state coverage, and application activity coverage, so as to optimize the rates of code coverage specifically for SARSA-generated test cases. Additionally, test suite prioritization techniques are introduced based on UI element coverage, test case cost, and test case complexity to further enhance the ordering of SARSA-generated test cases. Empirical investigations demonstrate that applying the proposed test suite prioritization techniques to the test suites generated by the reinforcement learning algorithm SARSA improved the rates of code coverage over original orderings and random orderings of test cases.

ACKNOWLEDGMENT

First and foremost, I want to express my deepest gratitude to my mother for her lifelong support. Her love and sacrifices paved my path to start my Ph.D.

I want to thank my wife for her love, patience, and trust in my abilities. Her encouragement has been the utmost motivation behind my achievements, which has assisted me in navigating the challenges of pursuing a Ph.D. degree.

I want to thank my Ph.D. supervisor, Renée Bryce, for her expertise, support, and guidance, for being such a nice person, and for giving me the opportunity to work with her. Her expertise in software testing and insightful feedback helped me formulate my research methodology.

I want to extend my gratitude to my committee members, Hyunsook Do, Barrett Bryant, and Stephanie Ludi. Their profound knowledge and constructive feedback helped me grow as a researcher.

I would like to acknowledge the support of my colleagues Shraddha Piparia, Ryan Michaels, David Adamo, and Farhan Rahman Arnob from the University of North Texas (UNT) Research Innovation in Software Engineering (RISE) laboratory. Their help, insightful suggestions, and contributions were invaluable to my research.

I am grateful to the Department of Computer Science and Engineering at the University of North Texas for the financial support I received to continue my doctoral studies.

Finally, I want to thank the University of North Texas Division of International Affairs and all my friends at UNT for making my Ph.D. journey eventful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

USE OF PREVIOUSLY PUBLISHED MATERIAL

The following articles have been reproduced, in whole or in part, in the sections indicated: David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. "Reinforcement learning for Android GUI testing." In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST 2018), pp. 2–8. https://doi.org/10.1145/3278186.3278187.

Reproduced with permission from the Association for Computing Machinery. Material is found in Sections 2.3, 3.1, 3.2, 3.3, 6.1 & 6.2; and Chapter 7

Md Khorrom Khan and Renée Bryce. 2022. "Android GUI test generation with SARSA." In 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0487-0493, https://ieeexplore.ieee.org/document/9720807.

Reproduced with permission from IEEE. Material is found in Sections 2.3, 3.1, 3.2, 3.4, 6.1 & 6.2; and Chapter 7.

Md Khorrom Khan, Ryan Michaels, Dylan Williams, Benjamin Dinal, Beril Gurkas, Austin Luloh and Renée Bryce. 2023. "Post prioritization techniques to improve code coverage for SARSA generated test cases." In 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC), pp. 1029-1035, https://ieeexplore.ieee.org/document/10099120.

Reproduced with permission from IEEE. Material is found in Sections 2.4, 6.1 & 6.2; and Chapters 4 & 7.

Md Khorrom Khan, Ryan Michaels, Farhan Rahman Arnob, and Renée Bryce. 2023. "Prioritization techniques for Android test suites generated by a reinforcement learning algorithm." Journal of Information and Software Technology (under review). Manuscript number: INFSOF-D-23-00290. Elsevier.

Material is found in Sections 2.4, 6.1 & 6.2; and Chapters 5 & 7.

CHAPTER 1

INTRODUCTION

The proliferation of Android applications in contemporary times has transformed how we interact with technology, making them an essential component of our everyday routine. Customization capability, the ability to install third-party applications, multitasking, external storage support, and fewer download and sharing restrictions than iOS are some of the many reasons that make Android the most popular smartphone operating system globally [113]. Android occupies almost 71.8% [113] of the mobile OS market share, with around 2.3 billion [93] users worldwide. Proper testing of an application before releasing it to the Play Store is crucial to prevent failure while using it and building confidence in it. An application failure can potentially be expensive and cause a loss of time, money, reputation, and even life. In 2020, substandard software quality led to a financial loss of $2.08 trillion (T) dollars in the United States [70]. According to "The App Attention Index 2021" [2], 57% of the users will not use a digital service if it cannot impress them in one shot, and 61% will not tolerate poor performance. Systematic and proper testing ensures the quality of the application and reduces the risk of failure at runtime. Due to the large state space, app creators are typically unable to test every possible scenario and must consider the cost-effectiveness of their testing processes. Manual testing poses significant challenges and lacks reliability due to the substantial investment of time and financial resources, combined with the inherent risks involved. Automated testing represents a valuable means of circumventing the shortcomings of manual testing, thereby resulting in enhanced productivity.

One of the most critical test automation tasks is automated test case generation. The intricacy of Android applications is compounded by the vast range of hardware configurations, an abundance of software versions, and multifaceted user interfaces, posing a distinctive challenge to developing automated test generation techniques. In contrast with conventional desktop applications that rely predominantly on keyboard and mouse inputs for user interactions, Android offers a range of input methods, including hardware and soft-

ware keyboards as well as touchscreens. The Android platform offers a diverse spectrum of gestures, including click, long-click, scroll, and rotate. It is imperative to acknowledge the interaction mechanisms and the distinctive framework architecture of Android when devising automated testing methodologies for Android applications.

Code coverage measurement is a critical aspect when evaluating the effectiveness of testing strategies. It evaluates the degree to which an application's code is executed during testing, encompassing both individual test cases and the entire test suite. Attaining high code coverage is highly important as it offers a valuable understanding of the thoroughness of the testing procedure by assessing the level of code coverage in the application's source code. This metric plays an essential role in assessing the effectiveness of the testing procedure and directly contributes to the identification and resolution of faults and vulnerabilities, ultimately enhancing the overall quality of the application. Applications that undergo extensive testing with high code coverage levels often demonstrate fewer flaws and higher reliability compared to those with lower coverage levels [66]. However, due to the complexity and diversity inherent in Android apps and devices, attaining satisfactory code coverage can present challenges, often requiring resource-intensive and time-consuming testing methods.

Graphical User Interfaces (GUIs) of mobile applications play an important role in providing the functionality end-users require, as end-users typically interact with an application through a GUI. By programmatically exploring the GUI, automated testing techniques can generate test cases that exercise different paths, inputs, and states of the application, leading to a more thorough examination of its behavior and increased code coverage. Over the past years, several researchers [19] [106] [110] [101] [31] [84] focused on developing automated GUI test-case generation techniques for Android applications. Numerous methods, including model-based, search-based, dynamic extraction-based, and machine learning-based techniques, have been utilized in these techniques to develop effective test scenarios for Android applications. In recent years, the widespread adoption of machine learning (ML) methodologies has significantly increased, resulting in its extensive utilization across multiple domains, including software testing. ML has garnered substantial momentum and is increasingly be-

ing embraced as an approach in the domain of software testing. One promising approach for automated test case generation in Android applications through GUI exploration to achieve high code coverage is the use of Reinforcement Learning (RL) [112]. Reinforcement learning is a machine learning approach that allows software agents to learn from their environment and make decisions based on the feedback received. By employing reinforcement learning algorithms, automated test case generation can be enhanced, enabling the test generation process to learn from the application's behavior and generate test cases that explore critical execution paths, thereby achieving a high degree of code coverage.

While automated test case generation is a significant advancement and achieves high code coverage, it often generates a large number of test cases, leading to time and resource constraints during execution. Additionally, the order in which these test cases are generated may not always be optimal. Consequently, optimizing the execution of these automatically generated test cases becomes crucial. Test Case Selection, Minimization, and Prioritization are popular test case optimization activities used in resource-limited scenarios. Selection ensures comprehensive coverage and optimized resource allocation, while Minimization eliminates redundancy. However, Prioritization is particularly significant as it efficiently allocates resources to address critical and high-risk test cases promptly, enhancing overall testing efficiency and effectiveness. Test Case Prioritization (TCP) schedules the test case execution order based on their importance, the likelihood of revealing faults, and the impact on critical functionalities. It is important to note that test case prioritization does not compromise the quality of testing since it does not discard any test cases. One of the key advantages of prioritizing test cases is its capability to improve the efficiency of code coverage. TCP improves the rate at which code coverage is attained by focusing on critical functionalities, complex user interactions, or previously untested areas of the application. It offers the capacity to sequence tests in a strategic fashion, predicated upon their relevance and potential for covering critical segments of the code, rather than executing them randomly or in the order they were created. As a result, fewer tests are required to achieve high levels of code coverage, saving both time and resources.

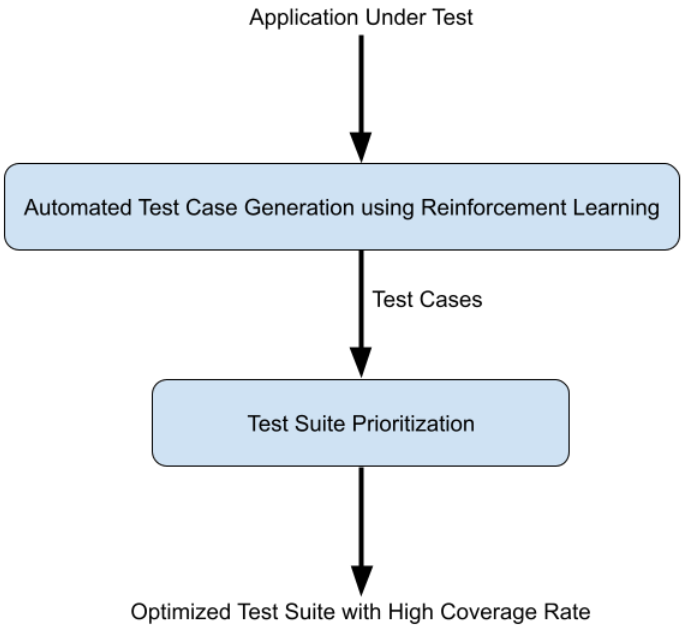


FIGURE 1.1. Overview of the Hybrid Testing Technique

1.1. A Hybrid Testing Approach

This dissertation presents a hybrid testing technique that integrates reinforcement learning-based automated test case generation with multi-perspective test case prioritization for Android applications. The objective of this integration is to achieve superior code coverage and optimize the effectiveness of the generated test cases by systematic reordering. The overview of the proposed hybrid approach is shown in figure 1.1. The application under test(AUT) first goes through a test generation process that utilizes reinforcement learning to systematically explore the AUT's GUI and thereby generating test cases as sequences of events. Then the test suite prioritization process reorders the test cases generated by reinforcement learning to their optimal order with the goal of maximizing the code coverage rate.

With the aforementioned objective, the dissertation adopts reinforcement learning algorithms, Q-learning and SARSA (State-Action-Reward-State-Action), to systematically generate test cases for Android applications with improved code coverage. Moreover, the dissertation presents test case prioritization techniques based on pairwise event interaction

coverage, application state coverage, and Android activity coverage to enhance the coverage rate of SARSA-generated test cases. A comprehensive empirical investigation uncovered a strong correlation between the attainment of a high code coverage rate in Android applications and the coverage of Android activities, which encompass individual screens containing UI (User Interface) elements accessible for user interactions. Guided by this observation, this study investigates the effect of UI element coverage on test case prioritization and develops novel prioritization strategies rooted in three key factors. These factors are the count of unique UI elements contained in the test case, the cumulative weight assigned to the test case, and the execution cost associated with it. The weight of a test case is determined by analyzing the distinct action types it encompasses while assigning varying weight values to each specific action type. To determine the cost of executing a test case, this study considers the test case length (number of events present in the test case) and the corresponding execution time measured in seconds. By leveraging these factors, this dissertation presents nine combinatorial strategies for calculating prioritization scores for each test case.

1.2. Contributions

This dissertation makes the following contributions:

- **Contribution 1(Published):** Adaptation of reinforcement learning algorithms, Q-learning and SARSA, for Android GUI test generation and empirical evaluation of the code coverage effectiveness of the generated test cases [12], [61].
- **Contribution 2(Published):** Developing test suite prioritization techniques based on pairwise event coverage, application state coverage, and activity coverage to improve code coverage for SARSA-generated Android test cases [63].
- **Contribution 3(Submitted to Information and Software Technology Journal, Current status is "Under Review"):** Developing test suite prioritization techniques based on GUI element coverage, test case cost, and test case complexity, measured in terms of "test case weight" for Android test suites generated by the reinforcement learning algorithm SARSA [62].

1.3. Dissertation Organization

The remainder of this dissertation is organized as follows:

Chapter 2 provides a comprehensive review of relevant literature aimed at establishing a foundational understanding of Android GUI testing. It explores existing approaches and techniques for test generation in the Android context, emphasizing the pivotal role of reinforcement learning algorithms. Additionally, it meticulously examines the landscape of test case prioritization techniques currently employed in the field.

Chapter 3 describes the details of the reinforcement learning setup and delves into the details of using reinforcement learning algorithms to generate effective test cases for Android applications. This chapter discusses the design and implementation of Q-learning and SARSA-based test case generation, highlighting their advantages over random testing.

Chapter 4 of this dissertation investigates prioritization techniques that improve the code coverage of test cases resulting from the SARSA reinforcement learning algorithm. It presents the utilization of pairwise interaction, activity, and application state coverage to improve the effectiveness of the generated test suite.

Chapter 5 focuses on element coverage and weighted cost-based prioritization techniques for Android test suites generated by the reinforcement learning algorithm SARSA. This chapter presents novel methods for prioritizing test cases based on their coverage of different application GUI elements, their associated costs, and their complexity, thus improving the efficiency and effectiveness of the generated test suite.

Chapter 6 summarizes the key findings in the context of Android GUI test generation using reinforcement learning algorithms and the prioritization techniques to optimize them.

Finally, Chapter 7 identifies opportunities for further research and improvement in the field of reinforcement learning-based Android GUI test generation and post-prioritization techniques to maximize testing efficiency, paving the way for future researchers to explore and expand upon the current work.

CHAPTER 2

BACKGROUND AND RELATED WORK

The Android Operating System (OS), originated by Google, is designed primarily for touchscreen mobile devices, including but not limited to smartphones and tablets. It has garnered widespread acclaim and emerged as the most widely utilized mobile operating system worldwide [113]. The open-source attribute of Android stands out as a salient feature. This characteristic empowers software engineers to adjust and tailor the operating system to align with their individual requirements. The adaptable nature of Android OS has proven instrumental in facilitating the proliferation of diverse Android devices, characterized by an array of sizes, designs, and pricing structures. In contrast to the exclusive availability of iOS on Apple devices, Android has the capacity to operate on a variety of smartphones and tablets from various manufacturers. Android provides a highly intuitive user interface coupled with an extensive array of applications that are readily accessible through the Google Play Store.

2.1. Android Application Components

Android applications are constructed based on the various components encompassed within the Android development framework, thus distinguishing them from traditional desktop applications. The fundamental elements of an Android application are its application components which function as an entry point for the user or the system to enter the application. The main components of Android applications are [21]:

- **Activities:** Android applications are composed of activities. An activity is a single screen or window of an Android application that contains a collection of widgets and events. Activities provide the foundation of the user interface for end-users and serve as the point of entry for user interactions, handle user input, and facilitate communication between different components.
- **Services:** Services refer to the background elements that execute long-running tasks or operations without any user interface.

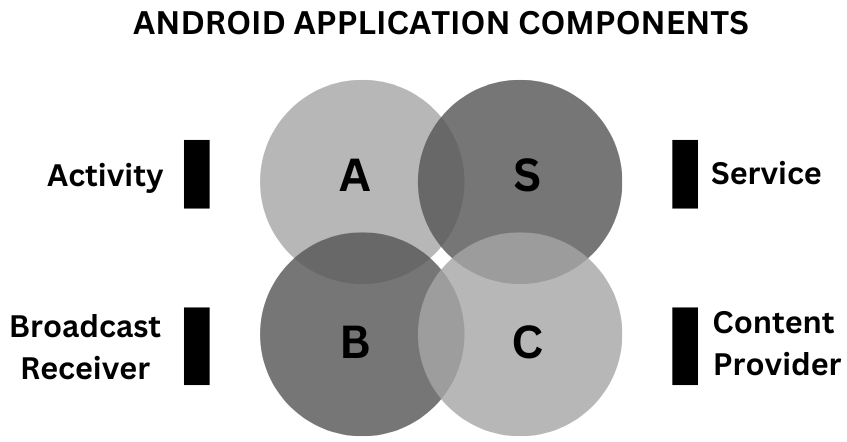


FIGURE 2.1. Android Application Components

They operate autonomously regardless of ongoing activities and persist in operation even in instances where the user transitions to a separate application. Services are typically employed to undertake various tasks, such as facilitating music playback in the background, managing network operations, or executing periodic data synchronization.

- **Broadcast Receivers:** Broadcast receivers allow apps to receive and respond to system-wide or application-specific broadcast messages. These messages may come from the system itself, such as low battery alarms, or they may be sent by other applications installed on the same device.
- **Content Providers:** Content providers facilitate the communication of data across applications. They control access to an organized set of data, such as a database or file, and offer other programs a standardized interface for querying or changing the data.

Effective testing methodologies must account for the unique architecture of the Android framework, particularly the activities, services, broadcast receivers, and content providers, in order to ascertain complete coverage of application functionality.

The official integrated development environment for Android uses either Java or Kotlin as the programming language to develop native Android applications. Cross-platform

application development frameworks such as Iconic [7], React [9], and Flutter [6] use programming languages including but not limited to Python, JavaScript, and C++. Even though native Android applications are mostly written in Java, we cannot use traditional tools for testing standalone Java applications for Android. Android Applications differ significantly and manifest different types of bugs [33]. The unique architecture of Android combines and extends concepts from multiple application domains, including embedded, web-based, distributed, and desktop [100]. Furthermore, the Android framework allows applications to support multiple input methods, such as hardware keyboards, software keyboards, and touchscreens. It offers a wide array of user interactions, including gestures like click, long-click, scroll, and rotate. These input methods and gestures need to be considered when developing test generation tools for Android applications. Dynamic UI management, system-generated events, inter-app communications, and context awareness [94] [45] are some of the other major challenges for Android test automation [100].

2.2. Software Testing

Software testing is a methodical and analytical procedure aimed at assessing a software system or application to ensure that it satisfies predetermined specifications, performs as designed, and functions consistently. The assessment of software quality, functionality, and performance is deemed a crucial aspect of the software development life cycle(SDLC) as it helps acquire trust in the product's dependability and performance, improving overall quality and user satisfaction.

2.2.1. Manual Testing

Manual testing is the process of manually running test cases and confirming software functioning without the use of automated tools or scripts. Testers engage directly with the program to do different test operations such as generating test cases, performing tests, and recording faults. To find problems, analyze software behavior, and certify compliance with requirements, manual testing depends on human observation, intuition, and knowledge.

2.2.2. Automated Testing

Automated testing entails the use of software tools, frameworks, and scripts to automate the generation and execution of test cases, and the verification of program functioning. It seeks to boost testing efficiency, expand test coverage, and decrease human effort in repetitive and time-consuming testing jobs. Automated testing tools enable testers to generate test cases using scripts or graphical interfaces, which are then run automatically by the tools. This method allows for faster test execution, fewer human mistakes, and more thorough test coverage.

2.2.3. Benefits of Automated Testing

The utilization of automated testing confers several advantages compared to manual testing.

- **Improved Testing Efficiency:** Automated testing tools are capable of executing tests at a significantly faster rate than manual testing techniques, thereby enabling prompt feedback on the software's performance while simultaneously shortening the overall duration of the testing process.
- **Increased Test Coverage:** The utilization of automation facilitates the implementation of a wide array of test cases and scenarios, encompassing a significantly wider scope of functionality, inputs, and configurations that may be infeasible to attain through manual testing.
- **Reduced Human Errors:** Automated tests are performed by automated tools, which eliminates the chance of human mistakes such as overlooking faults or inconsistencies, assuring consistent and accurate test execution.
- **Cost and Time Savings:** Although there may be a required investment necessary for the initial setup and maintenance of automated tests, the long-term benefits prove to be advantageous in terms of improved testing efficiency, reduction of manual labor, and early detection of defects during the development cycle, resulting in significant cost and time savings.

- **Parallel Testing:** Automated testing facilitates the concurrent execution of tests, thereby expediting the testing procedure to a significant extent. Organizations can enhance test execution time and overall efficiency by leveraging parallel testing via simultaneous testing across diverse devices or environments.
- **Scalability and Reusability:** Automated testing scripts and frameworks possess the ability to be conveniently scaled and utilized across diverse projects, platforms, and configurations. Once automated tests are developed, they can be utilized for subsequent releases or updates, reducing the time and resources required for generating fresh test scenarios. The scalability and reusability of testing practices are key factors in augmenting the efficacy and productivity of the testing process.

Through the incorporation of automated testing, it is possible to augment test coverage while enhancing software quality and reducing the overall release timeline. Ultimately, such implementation leads to the delivery of software products that are reliable and resilient, thereby meeting the expectations of the end-users.

2.3. Android Automated Test Generation

Android applications are Event Driven Systems (EDSs) where Graphical User Interfaces (GUI) have a prominent role. The GUI of an application is tightly coupled with the business logic to provide functionalities to the users. Testing application functionality through the GUI is critical. Many previous research efforts focused on Android testing through the GUI [19] [106] [110] [101] [31]. Manual testing can be used to test the Android application's GUI, but it's time-consuming. Android applications are composed of activities, and activities can have multiple widgets. The available events and their combinations require a significant amount of time and human effort. Automated testing may save time and cost and improve test coverage and efficiency. Automated test generation attempts to improve the efficiency and effectiveness of testing by generating test cases that cover a wide range of scenarios. Test generation for Android applications can be challenging due to the complexity of the user interface and the many devices and configurations on which Android apps can be used [100]. Characteristics of the Application Under Test (AUT) may influence

results as well. To be effective, test generation for Android must understand the UI and be able to generate test cases that exercise it effectively.

There has been ongoing research for years towards developing Android test automation tools [33] [128] [114] [59] [88]. Some of the common approaches used in the literature to develop automated testing tools for Android are discussed below:

2.3.1. Random Testing Tool: Monkey

A simple approach to automatically testing Android applications is to feed random GUI events to the Application Under Test (AUT) during its execution. The Android UI/Application Exerciser Monkey [1] is a command-line utility packaged with the official Android Software Development Kit (SDK). Monkey effectively injects a pseudo-random sequence of events based on screen coordinates, including touches, clicks, gestures, and some system-level events. It is important to note that, despite its popularity, Monkey lacks the ability to generate events by identifying the application's GUI elements; rather, it generates events by randomly clicking screen coordinates. As a result, Monkey often generates the same event multiple times or clicks on a non-interactive screen area. Sometimes it takes significant time for Monkey to reach the functionalities of the application, which, depending on testing resources, may result in subpar testing coverage. Haoyin [49] proposes optimization techniques to improve the fault detection capability of Monkey and presents a random walk-based tool that effectively comprehends the UI architecture of the application and uniformly disperses the event sequences across the entire input domain. Hu C et al. [52] combine automatic event and test case generation using Monkey with dynamic analysis of log files. This approach lacks accuracy as a significant amount of time is spent generating events that may not expose any faults or critical functionality. Amalfitano et al. [17] utilize the notion of the "Saturation Effect" by conducting concurrent random testing sessions in order to establish termination criteria for the Monkey fuzzing technique. Their methodology involves regular evaluations of the variation in code coverage among the ongoing sessions, ultimately ceasing the testing procedure upon identification of a deviation lower than a predetermined critical threshold.

2.3.2. Model-Based Testing (MBT)

Model-Based Testing (MBT) tools [25] [41] [111] [46] [72] [121] [122] [123] create an abstract model of the application under test (AUT) that describes the behavior of the AUT in terms of input sequences, actions, conditions, output, and the flow of data from input to output. MBT tools use the model of the application created as finite state machines or state charts to generate test cases.

Amalfitano et al. [18] presents a tool called *AndroidRipper* that dynamically constructs a model of the application under test as a GUI tree. Then it selects paths through the tree to generate test cases. Espada et al. [41] compose specially designed state machines to model the behaviors of an Android application. A model checker generates execution traces from the model corresponding to test cases. Choi et al. [32] use active learning with testing to learn a model of the application. Their main focus is to increase code coverage quickly and avoid restarting the application. MobiGUITAR [19] constructs the application's state machine by dynamically analyzing its GUI, followed by the application of pairwise edge (event) coverage criteria to generate test cases from the state machine. Stoat [111] employs a dynamic analysis approach, supplemented by weighted UI exploration and static analysis, to effectively study app behaviors and develop models, which are then repeatedly mutated and refined using Gibbs sampling [20]. The test case generation process utilizes a probabilistic approach to generate event sequences from the stochastic model. Gu et al. [47] abstract a model of the application under test and dynamically enhance the accuracy of the model by utilizing runtime information during the testing process. Their exploration strategy to generate test cases employs a combination of random and greedy approaches in the depth-first search.

One major drawback of model-based approaches is that they have a high tendency to generate infeasible test cases because even a simple application model with few states and transitions can generate a large number of event sequences and combinations, resulting in redundant and ineffective bug detection. Moreover, creating an accurate model takes time and can be error-prone because of the complexity of Android applications.

2.3.3. Search-based Techniques

Search-based techniques [16] [23] [106] [80] use meta-heuristic search optimization to generate test cases. EvoDroid [78] examines the source code of the application under test to create two different types of application models that describe the app's external interfaces and internal behaviors. To maximize code coverage, EvoDroid applies a step-wise evolutionary algorithm utilizing these two models. Amalfitano et al. [16] present AGRippin (Android Genetic Ripping), a Search-Based Testing approach that builds a GUI tree using a hill-climbing algorithm and then explores it using a genetic algorithm to generate test cases. SAPIENZ [80], a multi-objective search-based testing tool combines random fuzzing and systematic exploration with search-based exploration to detect faults and maximize coverage while minimizing the test case length. Jabbarvand et al. [57] use an evolutionary search technique to produce test cases by using a collection of application models reflecting both functional behavior and contextual variables that affect energy usage. Eler et al. [39] present MATE, an accessibility testing tool that uses a random strategy to explore the UI and searches for accessibility issues based on pre-configured accessibility properties whenever a screen state is found. Auer et al. [23] improve the performance of MATE [39] by integrating a surrogate model—an abstraction of the state-based behavior of GUI. This model can avoid executing high-cost test cases by tracing already-explored behavior. The Cadage framework, proposed by Zhu et al. [131], automatically constructs and refines a dynamic GUI model of the Android application under test. Then, it employs a breadth-first search algorithm with the objective of efficiently exploring unexecuted GUI event handlers, thereby ensuring thorough coverage during the testing process. Jha et al. [58] presents a value-deterministic search-based strategy that captures events and associated data during the execution of an Android application and stores them in a log file. Utilizing the log record, the technique applies a value-deterministic search-based replay to reproduce crashes.

2.3.4. Dynamic Analysis-Based Techniques

Dynamic extraction techniques [77] [120] generate and execute event sequences on the fly at run-time. They do not need an abstract model of the AUT to generate test cases. Choi

et al. [32] combine active learning with testing and propose a machine learning approach called Swifthand to learn the AUT dynamically. Their goal is to generate test cases to maximize branch coverage with minimal restarts. MacHiry et al. [77] present Dynodroid, which uses a dynamic approach to generate a sequence of events using an "observe-select-execute" cycle. It observes the available events first, confirms their existence, and then chooses using a randomized algorithm. Wu et al. [120] present a dynamic malware detection framework called DroidDolphin by leveraging GUI-based testing, big data analysis, and machine learning. It utilizes a support vector machine (SVM) algorithm to predict malware by analyzing API call logs and Android Virtual Device (AVD) logs. A3E, presented by Azim et al. [24], incorporates a dynamic depth-first exploration strategy that capitalizes on automated techniques to explore activities and graphical user interface (GUI) elements in a systematic way with the goal of maximizing activity and method coverage. ENCK et al. [40] created TaintDroid that tracks the stream of privacy-sensitive information through third-party applications utilizing dynamic taint analysis [105] to monitor sensitive data. PREFTEST [76] integrates static and dynamic analysis methodologies to comprehensively evaluate the effects of varied preference configurations on Android applications. Wen et al. [118] developed a fully distributed framework called PATS (Parallel Android Testing System) that dynamically analyzes the GUI of an application to enhance testing efficiency by parallelizing the testing process and identifying and eliminating redundant sequences. Alzaylaee et al. [15] introduced a hybrid testing approach, employing the Monkey random testing tool in conjunction with the state-based dynamic analysis tool DroidBot [73]. The primary aim of this approach is to enhance code coverage and, consequently, reveal possible malicious behaviors.

2.3.5. Machine Learning-based Techniques

Machine learning(ML) is a division of Artificial Intelligence(AI) that concentrates on creating algorithms and models adept at analyzing information, drawing conclusions, and making decisions without explicit programming. There has been a surge in the utilization of machine-learning methodologies to solve complex problems including software

testing challenges for Android. Rosenfeld et al. [98] used machine learning to categorize Android activities into seven distinctive types in their study. They relied on the notion that activities with comparable interface structures have similar characteristics. Following that, the researchers ran customized testing for each activity type, concentrating on the intended behavior and structure to find functional bugs. AppFlow [53], introduced by Hu et al. utilizes machine learning to automatically identify typical screens and widgets and empowers developers to create a library of modular tests for the primary function of an app category(e.g. checkout feature of an e-commerce app). The utilization of this library in new applications can alleviate the burden of performing manual smoke testing. Peng et al. [92] present MUBot (Multi-modal User Bot) that utilizes a multi-modal deep learning framework to acquire knowledge from interactive traces. The acquired knowledge allows MUBot to emulate human users and facilitate the automated test generation for intricate commercial Android applications. White et al. [119] propose a method to enhance random GUI testing through the utilization of a machine learning approach known as "You Only Look Once" (YOLO) for the automatic identification of GUI widgets in screenshots. The initial step involved training the machine learning model to identify the types and respective positions of widgets exhibited on the screen. The resultant outputs subsequently guide the test generator to generate test cases with improved coverage.

One machine learning technique that is receiving particular attention in the field of automated test generation is reinforcement learning. Reinforcement learning is a type of machine learning where algorithms learn to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. The learning agent explores the environment through a trial-and-error approach to maximize cumulative reward. Popular reinforcement learning algorithms include Q-learning, SARSA, Deep Q-Learning etc. AutoBlackTest [82] is perhaps the first Q-learning-based GUI testing solution for Java/Swing desktop applications. Esparcia-Alcázar et al. [42] propose TESTAR, a GUI testing tool based on Q-learning for desktop and web applications. Kim et al. [65] generate test cases with improved branch coverage for C applications using deep reinforcement learning with

their tool Gunpowder. These tools are not applicable for mobile testing because of the unique characteristics of mobile applications, such as a wide range of screen sizes, OS versions, input methods, and interaction mechanisms.

This work implemented an Android GUI testing strategy using Q-learning to increase code coverage [12]. The optimized event selection using trial-and-error interactions employed in the proposed technique accomplishes 3.31% to 18.83% better code coverage than random testing. Vuong et al. [116] propose a similar test generation algorithm based on Q-learning for Android with a different reward function than the reward function utilized in this study. In their approach, the discount factor was fixed at 0.9. Contrarily, the proposed technique in this study utilizes a variable discount factor based on the number of events in the subsequent state after executing an event, allowing the agent to look further ahead if a state has fewer events. QBE [69] explores GUIs and prioritizes the GUI action transitions using Q-learning to increase activity coverage and crash detection. Q-testing [90] is another Q-learning-based tool that uses a curiosity-driven exploration strategy and a neural network to distinguish functional scenarios. Yasin et al. [124] present DroidbotX, which increases overall instruction coverage, method coverage, and activity coverage by employing Q-Learning with upper confidence bound (UCB) exploration. Andrea et al. [97] use a deep neural network in their tool ARES to achieve higher code coverage and detect faults in Android apps. DeepGUIT [34] applies Deep Q-Network for Android testing. This approach uses a neural network to approximate the action-value function using current information(states, actions, rewards, and following states).

The model-based tool AIMDroid [46] is the sole tool found in the literature that uses the reinforcement learning algorithm SARSA for Android testing. Using a Breadth-first search (BFS) algorithm, AIMDroid traverses activities and creates a BFS tree. Then it encases an explored activity in a "cage" and employs a SARSA-guided fuzzing algorithm to explore the inner states of the activity. The primary goal of AIMDroid is to explore every activity and reduce activity transition time. This dissertation presents an Android GUI test generation based on SARSA to explore the UI space of the entire AUT instead of focusing on

only one activity. Similar to the Q-learning algorithm introduced in this study, the SARSA-based test generation technique systematically explores the application's unexplored areas using SARSA with the goal of maximizing code coverage without the need to create any BFS tree. The detailed description of the Q-learning and SARSA-based test generation algorithms implemented in this study is discussed in Chapter 3. The SARSA test generation provided increases of 9.87-24.79% code coverage, 6.90 - 20.09% branch coverage, 7.88 - 28.48% method coverage, and 3.74 - 35.02% class coverage when compared to the random testing tool Monkey [61]. While reinforcement learning-based test case generation through the SARSA algorithm significantly improved code coverage over random testing, there is still room for improvement in the ordering of the generated test cases. It is conceivable that the high-impact test cases are generated late in the process, potentially reducing the overall effectiveness of the generated test suite. Test case prioritization can improve the effectiveness of these test suites by scheduling test cases in the most optimal order.

2.4. Test Case Prioritization

Test case prioritization identifies the optimal order to execute the test cases based on predefined criteria, with the objective of executing the most critical test cases first. This technique can be especially beneficial when there are time constraints on testing, as it prioritizes the execution of critical test cases at the beginning of the test suite. Rothermel et al. [99] defined the test suite prioritization problem as:

DEFINITION 2.1. Find $T' \in PT$ such that $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$.

PT represents the possible orderings of an individual test case within a test suite T while the function f represents the prioritization fitness and assigns a prioritization score.

Test case prioritization has been the subject of extensive research, and a variety of techniques and algorithms [87] [36] have been developed based on different criteria including code coverage analysis [54], genetic algorithms [26], fault history analysis [89], interaction coverage analysis [27] and machine learning-based approaches [64].

2.4.1. Coverage-based Prioritization Techniques

Coverage-based prioritization techniques prioritize test cases based on their ability to cover specific aspects of the application. These aspects can include but are not limited to, code coverage [99] [37] [48], requirement coverage [96] [102] [129] [81] and combinatorial interaction coverage [27] [103] [28].

Rothermel et al. [99] presented several prioritization techniques that leverage statement and branch coverage to reorder test cases. The techniques were evaluated on eight C programs and found to be effective in quickly detecting faults. Jones et al. [60] introduced prioritization and reduction techniques for Modified Condition(MC)/Decision Coverage(DC) test suites used in commercial airborne systems. MC/DC is a stringent variation of the basic condition/decision coverage criterion that aims to ensure that each condition in a decision statement independently affects the decision outcome. The prioritization algorithm utilizes MC/DC pair coverage and an additional approach that recomputes the contribution of test cases after each test case is selected. Fang et al. [43] focused on logic coverage methods such as Branch coverage, and MC/DC coverage to prioritize test cases. Experiments imply that logic coverage is appropriate when fine-grained coverage information is available. Krishnamoorthi et al. [96] propose a requirement coverage-based prioritization strategy using six factors to improve fault detection rate. Konsaard et al. [67] employed a genetic algorithm to develop total coverage-based prioritization.

Bryce et al. [27] prioritized test cases for Event-drive Software(EDS) based on t-way combinatorial coverage of interactions. Huang et al. [55] present a novel approach to prioritization relying on fixed-strength interaction coverage. Their approach involves utilizing base choice coverage iteratively and implementing one-wise coverage strength as a means to boost the fault detection efficacy while ensuring the cost-effectiveness of the prioritization process. Satish et al. [104] proposed a hybrid technique that generates test cases using the ACTS tool [127] with strength 2, modifies the covering array by replacing "don't care" positions, and prioritizes test cases based on a cost function incorporating higher-order coverage.

2.4.2. Cost-aware Prioritization Techniques

Cost-based prioritization takes into account the expenses associated with executing a test case. Test cases are assessed not just for their ability to detect errors or vulnerabilities, but also for the resources necessary to carry them out. Various factors, including the execution time and the length of the test case, are considered in determining the associated costs. Malishevsky et al. [79] presented a novel cost-benefit model for test case prioritization that divides costs into the cost of analysis and the cost of the prioritization algorithm. Srikanth et al. [107] suggested a cost-effective prioritization method for configurable software systems. Their method uses configuration switching cost as a prioritizing element, giving low-cost tests more importance. Elbaum et al. [38] introduced a novel metric to assess the effectiveness of fault detection. This metric accounts for test case costs and the severity of detected faults, thereby offering a more comprehensive evaluation of the fault detection rate. Furthermore, they presented techniques aimed at prioritizing factors that consider such expenses. In their study, Bryce et al. [29] introduced a new prioritization algorithm that incorporates the coverage of pairwise interactions along with the test case length as its cost. The algorithm computes a test case's prioritization score by dividing the number of yet-to-be-covered pairwise interactions by the length of the test case. Hyuncheol et al. [91] and Huang et al. [56] used historical data of test case execution to prioritize test cases, utilizing a cost metric that takes into account both the severity of faults and the time taken for executing the test cases. Hyuncheol et al. calculated a "historical value" for each test case based on its fault severity and the execution time of the test case, while Huang et al. employed a genetic algorithm that takes into account the test case execution time and fault severity to determine the optimal order of test case execution.

2.4.3. Risk-based Prioritization Techniques

The goal of risk-aware test case prioritization is to rank test cases based on their possible influence on system dependability and functioning. It understands that not all test cases are created equal and that certain tests have a larger risk of creating major failures or vulnerabilities. Test cases are prioritized in risk-aware prioritization based on their associated

risks, such as the possibility of discovering a defect, the severity of the possible failure, and the criticality of the impacted functionality. Stallbaum et al. [109] presented RiteDAP (Risk Based Test Case Derivation And Prioritization), a technique for risk-based generation and prioritization of test cases that employs activity diagrams as its core mechanism to generate test cases. Then it prioritizes the test cases by their total risk score, as well as their additional risk score. The technique estimates risks by evaluating the probability of containing a fault by an entity and the total damage caused by this fault. Srivastava et al. [108] demonstrated a prioritization approach that considers requirement priority, severity, and the probability of risk factors associated with the requirements. To estimate risks, they followed a systematic approach: firstly, identifying potential problems, and subsequently assigning severity values to each identified problem. Yoon et al. [125] proposed a prioritization strategy utilizing product risks based on multiple risk items associated with the product. The suggested technique includes an initial assessment of risk weight associated with requirements, which is performed by evaluating the probability of failure and the cost paid as a result of the failure and multiplying the two together. The risk exposure values are then estimated based on the risk weight of requirements for various risk items. The prioritization technique systematically reorders test cases based on risk exposure values. Hettiarachchi et al. [51] employed a fuzzy expert system in order to assess the risks associated with project requirements and further recommended a simplified prioritization strategy predicated on the identified risks. The utilization of a fuzzy expert system is employed to derive the status of requirement modification and potential security vulnerabilities.

2.4.4. Test Case Prioritization for Android Applications

The majority of the literature on test case prioritization emphasizes desktop and web applications. The conventional methods of prioritizing test cases may not be suitable for testing mobile applications since they fail to account for the distinctive features of mobile devices. According to a study by Mukherjee et al. [87], there has been comparatively little research on test case prioritization in the mobile application domain. Marijan [83] introduced a prioritization strategy that takes into consideration various criteria such as the execution

time of tests, the frequency of test failures, the impact of test failures, and cross-functional implications in the context of a continuous integration environment. This was tested on three Android applications but did not factor in the core features of an application. The prioritization strategy introduced by Qian [95] et al. gives the highest priority to the test case which have the highest potential to trigger an Android memory leak. However, it does not take into account other factors such as code coverage or bugs that are not related to memory leaks. Waqar et al. [117] suggested an Android test case prioritization model using sequence patterns and reinforcement learning. An inaccurate RL model or the quality of the dataset collected from the users can affect the prioritized test suite's performance.

Michaels et al. [85] leveraged the graphical user interface (GUI) of Android applications, as well as the traits of automatically generated test cases, to develop test case prioritization techniques based on combinatorial coverage of element and event sequences. While these studies have provided valuable insights into test case prioritization for Android applications, there is still a need to investigate more effective and efficient approaches to achieve high code coverage and improve application quality. Additionally, there has been no prior research that has integrated test case prioritization alongside reinforcement learning-based test case generation for Android applications.

CHAPTER 3

REINFORCEMENT LEARNING FOR ANDROID GUI TEST GENERATION

This chapter introduces the fundamentals of reinforcement learning and provides a comprehensive overview of its application in the context of the Android environment to generate automated GUI tests. To begin with, the chapter delves into the essential concepts that form the foundation of reinforcement learning setup, with a particular focus on the crucial idea of balancing exploration and exploitation.

Moreover, the chapter provides an extensive exploration of the adaptation of reinforcement learning techniques to the Android environment, highlighting the intricate details involved. This encompasses a detailed discussion on state and event representation, the design of the reward function, the utilization of Q-value functions, and the definition of the discount factor.

Additionally, this chapter provides a thorough overview of the test case generation process utilizing Q-learning and SARSA. It encompasses several key elements, including algorithm overview, research questions, experimental setups, results analysis, discussions, and an exploration of potential threats to the validity of the findings.

3.1. Reinforcement Learning Fundamentals

Reinforcement Learning (RL) [112] [115] is a computational approach to machine learning, inspired by behavioral psychology and focused on goal-directed learning from interactions. In contrast to supervised learning, which involves the provision of labeled examples to an agent, and unsupervised learning, whereby an agent discovers patterns within unlabeled data, reinforcement learning is predicated upon a learning process reliant on trial and error. It directly connects an action with an outcome to learn about the best actions based on reward or punishment. Reinforcement Learning (RL) has the capacity to facilitate the creation of highly sophisticated systems that possess the ability to systematically learn and adjust to intricate settings, thus establishing a robust means of resolving an extensive array of practical problems.

The two main components of a Reinforcement Learning setup are the Agent and the Environment.

- **Agent:** The agent is an autonomous entity that can perform independent actions in an environment to achieve a goal. The agent interacts with the environment to learn and make informed decisions with the goal to learn an optimal policy that guides its decision-making process.
- **Environment:** The object where the agent is acting is considered to be the environment. It constitutes the external system or problem with which the Reinforcement Learning agent engages in interaction. It can be a virtual or physical environment, such as a game, a mobile application, a physical robot, or a financial market. The environment functions as a source of observations or states for the agent while also serving as the recipient of actions executed by the agent.

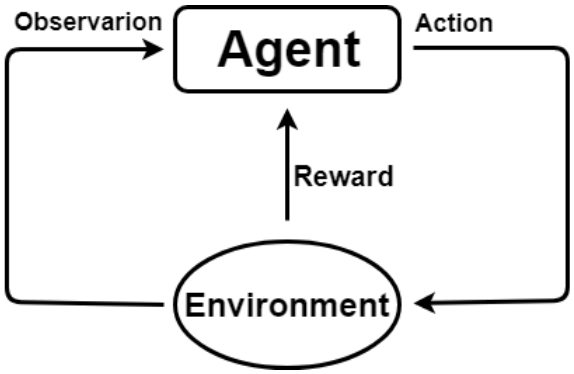


FIGURE 3.1. Reinforcement Learning

The utilization of trial-and-error interactions by the agent serves as a mechanism to procure knowledge pertaining to the environment and enhance its proficiency in decision-making. The decision-making process progresses sequentially in response to the interactions between the agent and the environment. In addition to the agent and the environment, there are other essential components in an RL setup:

- **State:** The state describes the current situation or configuration of the environment. The state comprises all the necessary information that the agent requires to make decisions. The state can incorporate numerous features such as the position of

objects, the agent's position, the current time step, or any other important factors. States can be discrete, with a limited number of potential states, or continuous, with an unlimited state space.

- **Action:** An action is a possible move or decision that an agent can take in a given state. In a certain state, typically an agent can perform finite actions. For example, the agent can move left, right, up or down in grid environment.
- **Reward:** The reward is an abstract concept to evaluate the actions. It is the immediate feedback from the environment after performing an action. The reward can be positive, negative, or even zero indicating the quality of an action.
- **Policy:** The policy is the strategy that the agent uses to select the next action from a certain state. The policy establishes a connection between the possible states and decisions of an agent, thereby directing the overall behavior of the agent.
- **Action-value function:** The Q-function, also referred to as the action-value function, is a critical notion that significantly influences the process of making decisions. The Q-function is employed to predict the aggregated rewards that will be acquired by an agent when taking an action within a defined state. The Q-function measures the quality or value of an action in a given state, steering the agent's decision-making toward interactions that are likely to result in higher long-term rewards. By iteratively updating the action-value function based on observed rewards, reinforcement learning algorithms such as Q-learning and SARSA can converge toward an optimal policy.

Figure 3.1 shows a typical reinforcement learning setup. In a reinforcement learning framework, an autonomous agent takes an action in a particular state based on a behavior policy, observes the resulting state, and collects the immediate reward for taking the action. Agent receives a positive reward for taking measurably good actions and a negative reward for measurably bad actions. From these observations, the agent updates the action value Q using a policy known as target policy or update policy that guides which action to take next in order to get the optimal cumulative reward. Popular reinforcement learning algorithms

include but are not limited to the Monte Carlo method, Q-learning, SARSA, Q-learning - Lambda, and SARSA - Lambda [112].

This study adopted two of the most popular reinforcement learning algorithms Q-learning and SARSA to generate test cases for Android applications where the application under test works as the environment.

3.1.1. Exploration and Exploitation

The fundamental concepts of exploration and exploitation hold significant importance in the reinforcement learning algorithm's learning process. Exploration is the process of searching out new and unexplored states and actions in the environment, whereas exploitation is the act of using learned information to make the best decisions based on present knowledge. Exploration is a crucial aspect of decision-making processes since it facilitates the collection of crucial information concerning the environment. Moreover, exploration provides the agent with the opportunity to identify and uncover hidden rewards, as well as explore alternative actions and states that potentially offer greater rewards. The absence of exploration may result in suboptimal solutions for the agent, leading to a failure in identifying the optimal policy. In contrast, the act of exploitation entails the utilization of existing knowledge to optimize immediate gains and benefits. Through the utilization of the learned policy, the agent is able to execute decisions that are anticipated to result in increased short-term rewards.

In reinforcement learning, balancing exploration and exploitation is a complex trade-off. Excessive exploration may result in excessive trial and error, wasting resources and time, whereas excessive exploitation may result in early convergence to unsatisfactory solutions. The optimization of exploration and exploitation during the process of learning presents a significant challenge.

3.2. Adaptation of Reinforcement Learning in Android Environment

This study employs the Q-learning and SARSA reinforcement learning algorithms to interact with an application through the iterative selection of Android GUI events from

various states within the application under test. The sequences of events generated as a result of GUI exploration are subsequently recorded and stored as test cases.

3.2.1. Representation of State and Event

In a particular state, the RL agent selects an event from the available events and evaluates the resulting reward based on the current state of the application. The state contains the information that helps the agent to determine the next event. The agent's job is to learn the sequence of events that maximize the cumulative reward.

The definition of what comprises a state and an event is crucial to the reinforcement learning-based approaches proposed in this study. The ability to deterministically identify each state and event enables test case generation to make event selection decisions across different iterations. During test generation, Appium [3] and UIAutomator [10] tools are used to retrieve XML (Extensible Markup Language) representations of an Android application's user interface. Using this XML representation, it is possible to discover the types of widgets available in a GUI state and the types of interactions (e.g. click, long press, etc.) that are enabled on the widgets. Widgets are uniquely identified by ID or XPath depending on what is available. This information provides the basis for the definition of state, action, and event.

Definition 3.1. An action a is denoted by a 3-tuple: $a = (w, t, v)$, where w is a widget on a particular screen, t is a type of action that can be performed on the widget (e.g. click) and v holds arbitrary text if the widget w is a text field. For all non-text field widgets, the value of v is empty.

Definition 3.2. A GUI state s is denoted by an n-tuple: $s=(a_1, a_2, a_3, ..., a_n)$ where a_i is an action and n is the total number of unique actions available on the screen.

Definition 3.3. An event is a set of one or more related actions that may occur in a particular GUI state. It is denoted by a 2-tuple $e=(s, A_e)$ where s is a GUI state and A_e is an ordered set of actions associated with the event.

In Android applications, a GUI widget can have multiple properties (e.g., position,

size, and label). Considering all these properties will create an enormously large number of states to test and cause state explosion. To manage the potential for state explosion, this study adopted a coarse definition of GUI state. As shown in definition 3.2, a GUI state is defined in terms of the events available on a particular screen. This is in contrast to considering the properties of every widget available on a screen.

3.2.2. Reward Function

The proposed approach models an Android application as a stochastic process with a finite set of GUI states S and a finite set of GUI events E (e.g., tapping a widget, text input, etc). In each GUI state s, the test generation system selects and executes an event e from the set of available events in s. Selecting and executing an event causes a transition to a new GUI state s'.

Event selection in a particular state is based on a notion of expected reward. The reward function computes the immediate outcome of carrying out an event. A definition of reward is necessary in order to enable the test generation system to distinguish between previously selected "good" and "bad" events. For instance, given a GUI state with text input fields that require specific types of input to move to a new screen, associating rewards to text input helps the test generation system re-select the input that led to new states in the past. The immediate reward $R(e, s, s')$ for executing event e in GUI state s is defined as:

$$R(e, s, s') = \begin{cases} r_{init}, & \text{if } x_e = 0 \\ \dfrac{1}{x_e}, & \text{otherwise} \end{cases} \tag{3.4}$$

where s' is the resulting state after executing event e, x_e is the execution frequency, i.e., the number of instances event e has occurred, and r_{init} is the default reward associated with each GUI event that has not yet been selected during the test generation process. Using this reward function, the events that have been explored previously are less likely to be explored again since the reward function is inversely proportional to the execution frequency. This helps ensure that a few events do not get selected too many times.

3.2.3. Q-value Function

The Q-value function determines the value of an event in a particular state of the AUT. It is based on the immediate reward for executing an event and the expected future reward associated with subsequent states. In other words, the choice of the event to select in a particular state is influenced not only by the immediate reward from selecting the event but also by potential rewards from events in future states. The Q-value function enables the test generation system to look ahead when making the choice of what event to select in a particular state. Sometimes immediate sacrifice may result in high future rewards in the long run. At each step, the agent selects a GUI event using an event selection policy guided by the Q-value of the available events in a given state of the AUT. The agent executes the event, observes the reward, and updates the action value Q using the Q-value function defined by the Bellman equation [112]:

Q-Learning:

$$Q(s,e) \leftarrow Q(s,e) + \alpha[R(e,s,s') + \gamma \cdot max_{e \in E_{s'}} Q(s',e^*) - Q(s,e)] \tag{3.5}$$

SARSA:

$$Q(s,e) \leftarrow Q(s,e) + \alpha[R(e,s,s') + \gamma Q(s',e') - Q(s,e)] \tag{3.6}$$

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

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

Q-Learning:

$$Q(s, e) = R(e, s, s') + \gamma \cdot max_{e \in E_{s'}} Q(s', e^*) \tag{3.7}$$

SARSA:

$$Q(s, e) = R(e, s, s') + \gamma Q(s', e') \tag{3.8}$$

The test generation system iteratively approximates the value of each GUI event in a particular state based on its experience by interacting with the application under test. The Q-value function enables the test generation agent to favor the execution of GUI events that lead to unexplored or partially explored states, irrespective of immediate reward. During the implementation, the Q-value of each event e is initialized to a user-defined default value. Each time an event e in a particular GUI state s is selected, the Q-value is updated using the Q-value equation.

3.2.4. Discount Factor

The discount factor helps the agent look ahead and determine how future rewards affect the Q-value function. For a state s' having a total number of events $|E|$, the discount factor $\gamma(s', E)$ is determined using equation 3.9.

$$\gamma(s', E) = 0.9 \times e^{-0.1 \times (|E|-1)} \tag{3.9}$$

Typically, the value of the discount factor is within the range of [0, 1]. A high discount factor encourages the test generation system to place a high priority on selecting events that lead to potentially high rewards in future states. A low discount factor specifies that the test generation system should focus on selecting events that maximize immediate reward. A discount factor of 0 makes the agent myopic as it considers only the immediate reward. On the other hand, a discount factor approaching 1 will always give priority to cumulative high future rewards. Instead of a static discount factor, this work utilizes a variable discount factor derived from the exponential decay function defined in equation 3.9. The intuition is that the agent should look further ahead (i.e. use a high discount value) and prioritize potential future rewards over immediate rewards when it encounters states with a small number of

events. The dynamic discount factor as defined in equation 3.9 reduces the tendency of the agent to ignore states that have a small number of available events.

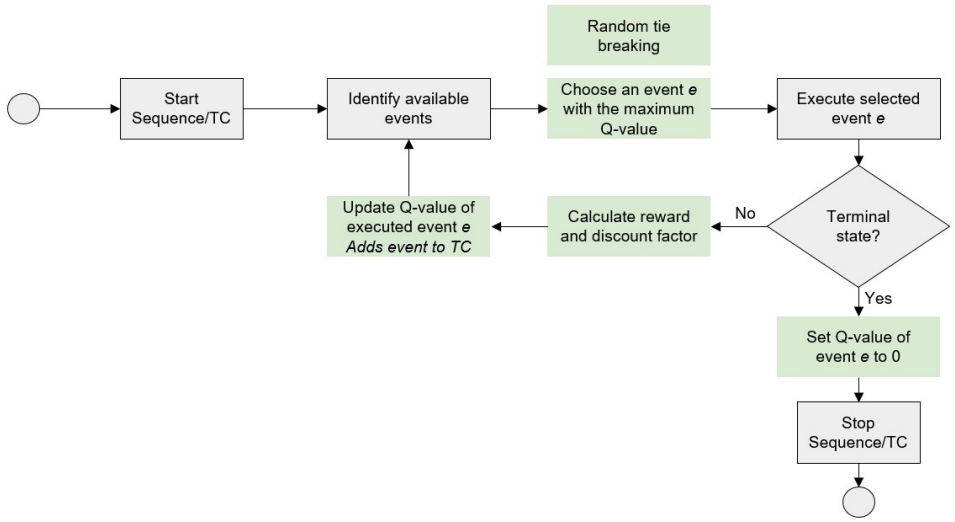


FIGURE 3.2. Test Case Generation with Q-learning

3.3. Android GUI Test Generation Using Q-Learning

The Q-learning-based test generation agent employed in this study adopts a greedy policy to carefully select events and subsequently, generates event sequences. In each state, the agent chooses an event that has the highest Q-value from the set of available events. Figure 3.2 shows the workflow of a test case generation process. The process begins by initializing the sequence and identifying the available events in the current state of the application. It chooses an event using a greedy policy, selecting the event with the highest Q-value from the available events. If two events have the same highest Q-value, one of them is randomly selected. The chosen event is executed, and the agent observes the resulting state. If it's not a terminal state, the test generation process calculates the reward and discount factor, updates the Q-value of executed event e, and adds the executed event to the sequence. This process continues until it finds a terminal event i.e. the event that causes application exit. When a terminal state is found, the algorithm updates the Q-value of the executed event to 0 and stops the sequence. The sequence of events created by this process is saved as a test case.

Algorithm 1: Test Suite Generation with Q-learning

input : application under test, AUT
input : test suite completion criterion, c
input : home button probability, $home\_btn\_prob$
input : initial Q-value, V_{init}
output: test suite, T

```
1  begin
2      while not c do
3          start AUT;
4          testCase ← φ;
5          while true do
6              if random(0, 1) <= home_btn_prob then
7                  selectedEvent ← HOME
8              else
9                  currEvents ← getAvailableEvents();
10                 foreach event in currEvents do
11                     if timesExecuted(event) = 0 then
12                         setQValue(event, V_init);
13                     end
14                 end
15                 selectedEvent ← getMaxValueEvent()
16             end
17             execute selectedEvent;
18             testCase ← testCase ∪ selectedEvent;
19             if selectedEvent exits AUT then
20                 updateReward(selectedEvent, 0);
21                 setQValue(selectedEvent, 0);
22                 break;
23             end
24             newEvents ← getAvailableEvents();
25             reward ← getReward(selectedEvent);
26             γ ← calDiscountFactor(newEvents);
27             maxValue ← getMaxValue(newEvents);
28             qValue ← reward + γ × maxValue;
29             setQValue(selectedEvent, qValue);
30         end
31         T ← T ∪ testCase;
32     end
33 end
```

3.3.1. Algorithm Overview

Algorithm 1 shows the pseudocode for the Q-learning-based test generation algorithm. It takes four input parameters:

(1) application under test,

(2) test suite completion criterion,

(3) probability of HOME button, and

(4) initial Q-value for new events.

The algorithm is part of an Android test generation tool called Autodroid [11] [13]. It uses the input parameters to explore the GUI and produces a set of event sequences as a test suite for the *AUT*.

The criterion for test suite completion is a fixed time budget. On each iteration, the algorithm creates an empty test case and starts the AUT. The *getAvailableEvents* procedure on line 9 retrieves all the events available in the current GUI state at the time it is called. Lines 10-14 set the initial Q-value to the user-specified value V_{init} for events that have never been executed. The *getMaxValueEvent* procedure on line 15 selects the event that has the maximum Q-value from the events in the current GUI state and line 17 executes the selected event. The call to *getAvailableEvents* on line 24 gets the available events in the new GUI state resulting from executing the selected event in the previous state. Lines 25-26 calculate the reward and discount factor for the executed event as defined in equations 3.4 and 3.9 respectively. The *getMaxValue* procedure on line 27 returns the maximum Q-value in the resulting state. Each time an event is executed, line 28 calculates the Q-value using equation 3.7 and line 29 updates it.

The event-selection and Q-value update process repeats until the agent executes a termination event that causes the AUT to close. The events executed until the termination point represent a single test case. Examples of termination events include:

(1) events whose sole purpose is to exit the application (e.g. clicking an exit button in a menu),

(2) pressing the HOME button,

(3) pressing the BACK button in certain GUI states,

(4) GUI events that cause a switch to some other application e.g. the Android contacts application and

(5) events that cause the AUT to crash.

Lines 19-23 assign a Q-value of zero to termination events. This enables the test generation agent to avoid previously encountered termination events that may prevent deeper exploration of the GUI in subsequent test cases. This blacklisting scheme also helps to prevent the generation of a high number of short test cases. It does not apply to the Android HOME button since the HOME button is used solely to probabilistically terminate each test case as shown on lines 6-7. Probabilistic termination of event sequences enables the agent to produce test cases of varying lengths within a test suite. An event sequence hash function is used to prevent the generation of duplicate test cases so that every test case in a test suite is unique.

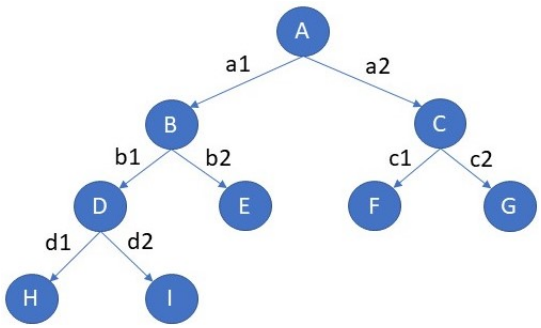


FIGURE 3.3. Example application in terms of states and events

3.3.2. Running Example

Consider the generation of a test suite for an Android application that has states A, B, C, D, E, F, G, H and I as shown in Figure 3.3. State A has two available events ($a1$ and $a2$), state B has two available events ($b1$ and $b2$) and state C has two available events ($c1$ and $c2$). States E, F, G, H and I are leaf nodes that represent the result of executing a termination event. The initial Q-value for each event is set to 500. The directed arrow indicates a transition from one state to another upon execution of the event indicated by the arrow's label. For instance, executing event $a1$ causes a state transition from A to B.

TABLE 3.1. Example rewards and Q-values for three episodes

state	event	Episode 0			Episode 1			Episode 2		
		Count	**Reward**	**Q-value**	**Count**	**Reward**	**Q-value**	**Count**	**Reward**	**Q-value**
A	a1	0	-	500	1	1	408	1	1	408
	a2	1	1	408	1	1	408	2	0.5	407.5
B	b1	0	-	500	0	-	500	0	-	500
	b2	0	-	500	1	0	0	1	0	0
C	c1	1	0	0	1	0	0	1	0	0
	c2	0	-	500	0	-	500	1	0	0
D	d1	0	-	500	0	-	500	0	-	500
	d2	0	-	500	0	-	500	0	-	500

As shown in Algorithm 1, app execution starts at line 3 and the number of events in the *testCase* is initially 0. This is considered as the start of an episode.

At the start of episode 0, the agent is in state A, and line 9 of the algorithm sets the *currEvents* value to {$a1$, $a2$}. Lines 10-14 set the Q-value for $a1$ and $a2$ to the user-specified initial Q-value of 500. Line 15 selects an event that has the highest Q-value in the current GUI state. Since both $a1$ and $a2$ have the same Q-value, one of them is selected randomly. If we assume that event $a2$ is selected, then line 17 executes the *selectedEvent* $a2$. This causes a transition from state A to state C and the execution count of $a2$ is updated to 1 (the initial execution count was 0). Line 18 adds the executed event $a2$ to the *testCase*. If the executed event closes the application, lines 19-23 set the reward and Q-value for the event to 0. Event $a2$ does not close the application, so line 24 sets the value of *newEvents* to {c1, c2}. Line 26 calculates the discount factor using equation 3.9. Line 25 calculates the reward for executing event $a2$ as defined in equation 3.4 and line 27 gets the maximum Q-value of the events $c1$ and $c2$ which is 500 (both events still have the same initial Q-value since they have never been executed). Lines 28-29 calculates and sets the new Q-value for the executed event $a2$. In state C, both events have the same Q-value. Suppose $c1$ is selected randomly and executed, then a transition is made from state C to F. The reward and Q-value for $c1$ are set to 0 since state F is a terminal state. The agent repeats this process for each episode until it executes a termination event that closes the AUT. Application exit indicates the end of an episode/test case and the resulting test case is added to the test suite. Table 3.1 shows the reward and Q-value for each event after each episode.

In episode 1, the agent uses information derived from episode 0. In state A, it selects and executes event $a1$ since it has the largest Q-value. This causes a transition from state A to state B. In state B, both $b1$ and $b2$ have the same Q-value. If we assume $b2$ is selected, then a transition from state B to E occurs. Since E is a terminal state, the reward and Q-value for $b2$ are set to 0 and the application closes. The updated reward and Q-value for each event in episode 1 are shown in Table 3.1 under the column "Episode 1". The algorithm continues to generate test cases until it meets the specified test suite completion criteria. This study employs a completion criteria of 2 hours time budget.

3.3.3. Research Question

The Q-learning-based technique is evaluated by comparing its performance to random test generation in terms of code coverage across eight subject applications. The goal is to answer the following research question:

Research Question: Does the Q-learning-based algorithm generate test cases with higher code coverage than random test generation?

3.3.4. Applications Under Test

The eight subject applications are downloaded from multiple categories in the F-droid [5] open source repository and they serve different purposes:

(1) Tomdroid
(2) Loaned
(3) Budget
(4) ATimeTracker
(5) Repay
(6) SimpleDo
(7) Moneybalance
(8) WhoHasMyStuff

Tomdroid is a note-taking application. *Budget* is an application to manage income and expenses. *ATimeTracker* helps users to start/stop time tracking for any task. *Moneybal-*

ance tracks expenses shared by groups of people. *WhoHasMyStuff* and *Loaned* are inventory apps to keep track of personal items. *Repay* is an app to keep track of debts. *SimpleDo* is a to-do list application.

TABLE 3.2. Characteristics of subject applications

App Name	**# Lines**	**# Methods**	**# Classes**	**# blocks**
Tomdroid v0.7.2	5736	496	131	22169
Loaned v1.0.2	2837	258	70	9781
Budget v4.0	3159	367	67	9129
ATimeTracker v0.23	1980	130	22	8351
Repay v1.6	2059	204	48	7124
SimpleDo v1.2.0	1259	88	31	5355
Moneybalance v1.0	1460	163	37	4959
WhoHasMyStuff v1.0.25	1026	90	24	3597

Table 3.2 shows characteristics of the subject applications including the number of lines, methods, classes, and bytecode blocks in each application. The applications range from 1026 to 5736 lines of code, 88 to 496 methods, 22 to 131 classes, and 3597 to 22169 bytecode blocks. *Tomdroid* has the largest number of code lines and bytecode blocks with 5736 and 22169 respectively. *WhoHasMyStuff* has the smallest number of code lines and bytecode blocks with 1096 and 3597 respectively. The applications contain a variety of input controls such as buttons, checkboxes, radio buttons, spinners, pickers, options menus, floating contextual menus, pop-up menus, and dialog boxes. The bytecode of each Android application was instrumented using the techniques described in Zhauniarovich et al. [130]

3.3.5. Experimental Setup

Random test generation is used as a baseline to evaluate the performance of the Q-learning test generation technique. Random test generation selects and executes events uniformly at random from the available events in each GUI state. The random and Q-learning-based test generation algorithms are implemented in the same tool, Autodroid [11], which is written in Java, to minimize the influence of different tool implementations on the results of the experiment. Autodroid takes instrumented APK files as input to generate test

suites and code coverage reports. Code coverage reports are generated using Emma [4]. The test cases were generated on Android 4.4 emulators with a screen resolution of 768x1280. The emulators were executed on a host machine running Ubuntu 14.04 with 16GB RAM. Table 3.3 shows the configuration parameters used to instantiate the random test generation and Q-learning-based algorithms.

TABLE 3.3. Test generation parameters

Parameters	**Random**	**Q-learning**
Generation time for each test suite (in hours)	2	2
Number of test suites (trials) for each app	10	10
Time delay between actions (in seconds)	4	4
Home button probability	0.05	0.05
Initial Q-value	-	500

Each test generation algorithm is run for 2 hours on each application to generate a test suite. Based on the size of the applications, 2 hours of testing time seemed reasonable for covering most of the application features. Both test generation algorithms are run 10 times on each subject application to minimize the impact of randomness in the algorithms. A time delay of 4 seconds between events was used so that the AUT has sufficient time to respond to one event before performing the next one. The probability of pressing the HOME button in a GUI state influences the average length of test cases in a test suite since the HOME button causes the AUT to exit. The HOME button probability was set to 5% since prior experiments suggest that the 5% probability value provides a reasonable balance between short and long test cases. For the Q-learning algorithm, a high initial Q-value of 500 was assigned to the events that have never been executed. This value is larger than any Q-value the test generation agent will derive from actual interaction with the AUT. Such a high initial Q-value encourages the test generation agent to execute each event in a GUI state at least once before making decisions based on learned Q-values. The high initial Q-value of 500 also encourages repeated execution of event sequences that revisit partially explored states (i.e. states with at least one event that has never been executed).

TABLE 3.4. Average block coverage achieved by the Random and Q- learning

App	Random	Q-learning	Improvement by Q-learning
Tomdroid	44.06%	**48.20%**	**4.14%**
Loaned	46.78%	**62.28%**	**15.50%**
Budget	65.95%	**69.26%**	**3.31%**
ATimeTracker	62.47%	**77.31%**	**14.84%**
Repay	44.79%	**55.69%**	**10.90%**
SimpleDo	50.41%	**69.24%**	**18.83%**
Moneybalance	78.90%	**87.50%**	**8.60%**
WhoHasMyStuff	76.50%	**82.75%**	**6.25%**

3.3.6. Results and Discussion

Code coverage is employed as a measure of the extent to which each test generation technique explores the functionality of the AUT (Application Under Test). Rather than comparing results separately for each application, the combined data across all applications is normalized and compared for each technique. In line with recommendations in Arcuri et al. [22], Mann-Whitney U-test is utilized for pairwise statistical tests to determine if there was a significant difference in block coverage between the Q-learning-based and random test generation techniques.

Table 3.4 shows the average block coverage achieved by random test generation and Q-learning-based technique. The Q-learning technique achieves higher average block coverage for all applications and shows an average improvement of 10.30% compared to random test generation. The difference between average block coverage achieved by the random and Q-learning-based techniques ranges from 3.31% to 18.83%.

Figure 3.4 shows a box plot of block coverage achieved by each algorithm for all the subject applications. The Q-learning-based technique has a higher median block coverage compared to random test generation for each of the applications. Q-learning consistently achieves higher maximum coverage than random test generation whereas random test generation always has the lowest coverage for each application. The Mann-Whitney U-test shows that there is a significant difference ($U = 1911.5, p = 0.00001$) between the block coverage of both techniques at the $p < 0.05$ significance level.

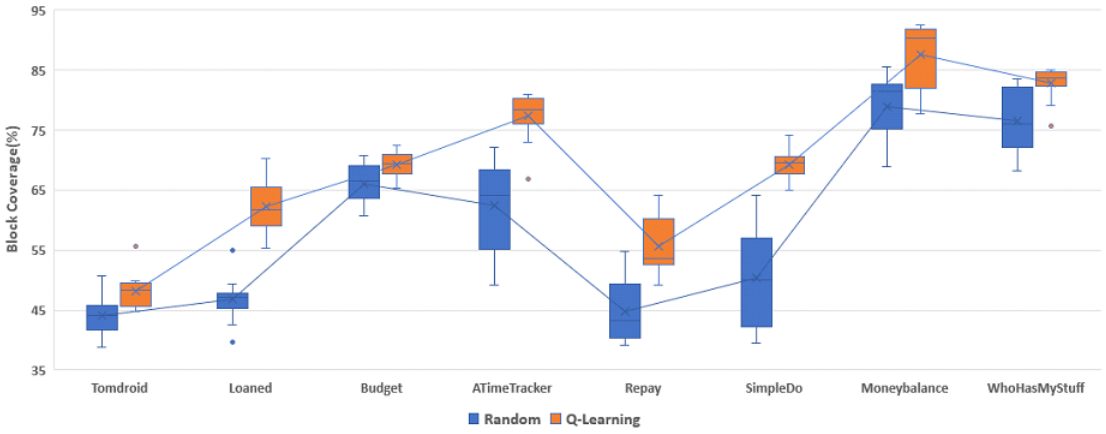


FIGURE 3.4. Block coverage across all applications and all runs

The difference in block coverage between the random and Q-learning-based techniques is most notable in *SimpleDo*, *Loaned*, *ATimeTracker* and *Repay*. These apps have GUIs that mostly require simple actions such as clicks and long presses rather than interactions with validated input fields. Furthermore, the majority of their functionality is accessible through a small subset of GUI states. The Q-learning-based algorithm assigns Q-values to encourage the execution of events that lead to new or partially explored states. This enables the algorithm to repeatedly execute sequences of high-value events and revisit the subset of GUI states that provide access to most of an AUT's functionality. The block coverage improvement in *Budget*, *Moneybalance*, *Tomdroid*, and *WhoHasMyStuff* is smaller than the other subject applications for a number of possible reasons. *Moneybalance* and *WhoHasMyStuff* have a small number of features, most of which are easily accessible by the random and Q-learning-based algorithms. *Budget* has several GUI states that require complex interactions with validated text input fields. *Tomdroid* has a significant amount of OS-specific and configuration-dependent code that is unreachable regardless of which test generation algorithm is used. The block coverage improvements in these apps may be due, in large part, to the Q-learning algorithm's ability to identify and avoid termination events that prevent deep exploration of the GUI within a single episode.

3.3.7. Threats to Validity

While Q-learning-based test generation has resulted in superior code coverage compared to random test generation, there are several potential threats to consider:

Internal Validity: One potential internal threat to validity present in this research is the selection of input parameter values for the Q-learning algorithm. The selection of parameters, including the discount factor, the heuristics used for event selection, and the initial Q-value, bears considerable influence on the behavior and effectiveness of the algorithm. Determining optimal parameter values that perform consistently across diverse scenarios poses a significant challenge, especially in the context of machine learning algorithms. The diversity in the selection of parameters has the potential to create partiality or irregularities in the process of test generation, thereby giving rise to possible discrepancies in the outcomes. The research employed a learning rate of 1 and integrated a variable discount factor determined by an exponential decay function. The selection of these values was founded on the intuition that the agent ought to show a forward-looking behavior by assigning higher discount values to states with fewer events.

External Validity: The study's sample size is limited, as it compared the Q-learning technique to random test generation across only eight open-source Android applications. The limited number of subject applications reduces the diversity and representativeness of the sample, thereby restricting the generalization of the results to a more comprehensive selection of Android applications. Furthermore, comparing only one alternative methodology limits the external validity because it does not give a full evaluation against a variety of competing methods. Further studies that compare the Q-learning-based technique to greedy frequency-based [11] and combinatorial-based [13] alternatives, with a higher number of subject applications, may increase confidence in the results. Nevertheless, the consistency of the results in this initial study suggests that our Q-learning-based technique has potential value.

Construct Validity The assessment of the Q-learning-inspired test generation method exclusively with regard to block coverage imposes a constraint that has the potential to impinge

upon the study's construct validity. To enhance construct validity, future research should investigate incorporating metrics and assessments that capture a more thorough picture of the technique's usefulness and performance. Further studies may examine fault detection ability and the impact of different time budgets for applications of different complexity.

3.4. Android GUI Test Generation Using SARSA

SARSA behaves and learns in accordance with the same policy and hence is known as an on-policy algorithm. In other words, SARSA uses the same policy to select an action and update the action value Q. At each step, the SARSA agent selects a GUI event from the available events in a given state from the AUT. The agent executes the event, observes the reward, and updates the action value Q using the Q-value function defined by the equation 3.8. Figure 3.5 shows the workflow of a test case generation process using SARSA. The process begins by initializing the sequence and identifying the available events in the current state of the application. It chooses an event using an ϵ-greedy policy from the available events. The chosen event is executed, and the agent observes the resulting state. The Q-value is set to 0 for terminal states. For non-terminal states, the agent selects a new event from the new state using the same ϵ-greedy policy. Then the Q-value of the executed event is calculated and updated based on the reward, discount factor, and the Q-value of the newly selected event. This process continues until a terminal state is reached.

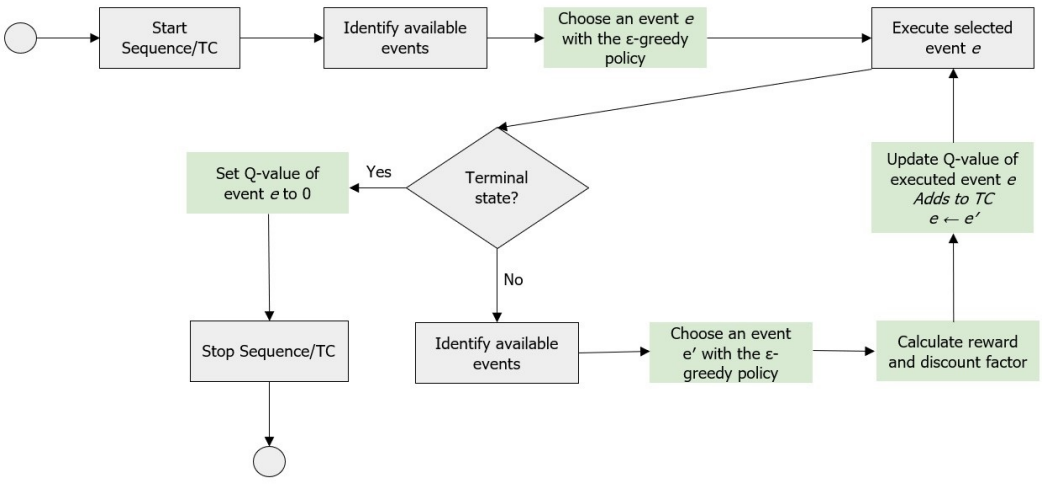


FIGURE 3.5. Test Case Generation with SARSA

Algorithm 2: ϵ-greedy event selection

Input: available events in a given state, *events*
Input: epsilon value, e
Output: selected event, *eventToExecute*

```
1 p ← random(0, 1);
2 if p < e then
3 |   eventToExecute ← selectRandom(events);
4 else
5 |   eventToExecute ← getMaxValueEvent(events);
6 end
7 return eventToExecute
```

3.4.1. Epsilon Greedy Event Selection

The ϵ-greedy exploration policy is utilized to select an event and update the Q-value. In this approach, the agent randomly selects an event with a probability of ϵ and the best event (the event with the maximum Q-value in a state) with a probability of 1-ϵ. The value of ϵ determines the classic problem of exploration vs. exploitation. The agent will select a random event most of the time if the ϵ value is high, and with a low ϵ value, the agent will exploit more, i.e., most of the time, the agent chooses a greedy event. This study hypothesizes that the ϵ value of 0.3 is a good exploration vs. exploitation trade-off considering the large exploration space of Android applications. Algorithm 2 shows the pseudocode for ϵ-greedy event selection. It is the definition of *getEpsilonGreedyEvent* used in the Algorithm 3 line 11 and 29.

3.4.2. Algorithm Overview

Algorithm 3 illustrates the pseudocode for the test generation process using SARSA. It accepts the AUT(Application Under Test), a test suite completion criterion c, initial Q-values V_{init}, and a test case termination criterion t as inputs and generates a test suite T as output. The test suite completion criterion is a predetermined time budget of two hours, and the test case termination criterion is the probability of selecting the home button set to 0.05, which is the same criterion used in Q-learning-based test generation. Initially, a high Q-value of 500 was set to ensure that the events get executed at least once.

Algorithm 3: Test Suite Generation with SARSA

input : AUT
input : completion criterion, c
input : initial Q-value, V_{init}
input : test case termination criterion, t
output: test suite, T

```
1  begin
2      while not c do
3          start AUT;
4          TC ← φ;
5          events ← getAvailableEvents();
6          foreach event in events do
7              if execFrequency(event) = 0 then
8                  setEventValue(event, V_init);
9              end
10         end
11         eventTemp ← getEpsilonGreedyEvent(events);
12         while not t do
13             eventToExecute ← eventTemp;
14             execute eventToExecute;
15             TC ← TC ∪ eventToExecute;
16             if eventToExecute exits AUT then
17                 updateReward(eventToExecute, 0);
18                 setEventValue(eventToExecute, 0);
19                 break;
20             end
21             newEvents ← getAvailableEvents();
22             foreach event in newEvents do
23                 if execFrequency(event) = 0 then
24                     setEventValue(event, V_init);
25                 end
26             end
27             γ ← calculateDiscountFactor(newEvents);
28             reward ← getReward(eventToExecute);
29             eventTemp ← getEpsilonGreedyEvent(newEvents);
30             eventValue ← getEventValue(eventTemp);
31             qValue ← reward + γ × eventValue;
32             setEventValue(eventToExecute, qValue);
33         end
34         T ← T ∪ TC;
35     end
36 end
```

On each iteration of the outer while loop, the algorithm initializes the AUT and sets the test case TC as an empty set. It then retrieves the available events in the current state of AUT using the function *getAvailableEvents* (line 5). Next, the algorithm initializes the Q-values of the available events (lines 6-10). It selects an event (line 11) using the ϵ-greedy event selection policy described in algorithm 2. Then it enters the inner while loop to create a test case i.e. sequence of events. Within the loop, the algorithm executes the selected event (line 14) and adds it to the test case TC (line 15). The algorithm observes the resulting state. If the event causes the AUT to exit, the event's reward Q-value is updated to zero (lines 16-20). The algorithm terminates the test case and starts a new one.

If the event does not cause the AUT to exit, the algorithm retrieves the newly available events (line 21) and initializes their Q-values (lines 22-26). It calculates the discount factor, gamma, based on the new events (line 27), retrieves the reward of the executed event (line 28), and selects a new event to execute using an epsilon-greedy policy (line 29). The Q-value of the executed event is updated based on the reward, discount factor, and the value of the selected event (lines 31-32). This process continues until the test case termination criteria are triggered. Once the termination criterion t is met, the test case TC is added to the test suite T (line 34). The algorithm continues the process until the completion criterion c is satisfied.

3.4.3. Research Questions

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

RQ1. **Is SARSA able to generate test cases with better code coverage than Monkey?**

RQ2. **How does SARSA compare to Monkey in terms of code coverage progress over time?**

TABLE 3.5. Characteristics of subject applications

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

3.4.4. Applications Under Test

Seven Android applications of various sizes and categories, downloaded from the open-source app repository F-droid [5], are utilized to evaluate the code coverage performance of SARSA. The applications are:

(1) AnkiDroid

(2) Tricky Tripper

(3) Track Work Time

(4) The Kana Quiz

(5) Tickmate

(6) SimpleReminder

(7) Open FNDDS Viewer

Table 3.5 shows the number of lines, branches, methods, and classes for each of the subject applications. The applications exhibit a variation in their code base size, with lines of code ranging from 971 to 29063, branches ranging from 163 to 11772, methods ranging from 189 to 4091, and classes ranging from 37 to 500. *AnkiDroid* has the highest, and *Open FNDDS Viewer* has the lowest number of lines, branches, methods, and classes. The applications are instrumented to collect code coverage using the free and open-source code coverage tool JaCoCo [8]. The test generation process installs the instrumented APK and runs the experiments. It uses Appium [3] and UIautomator [10] to extract the XML representation of the AUTs GUI and discover available widgets and actions identified by

unique ID or XPath.

3.4.5. Experimental Setup

The random test generation tool Monkey, which comes with Android Studio, is utilized as a baseline to assess the performance of code coverage performance of the SARSA-based test generation algorithm. Monkey provides high code coverage and has been used as a baseline in many research studies. Typically, Monkey generates a single sequence for a test suite, while the SARSA-based algorithm proposed in this study generates multiple event sequences. Therefore Monkey was configured to run for two hours and generate multiple event sequences for a fair comparison. The implementation of the SARSA algorithm and the reconfigured Monkey tool was conducted using Python programming language. The maximum event sequence length for each application generated by SARSA was used as the input for the Monkey event sequence. Monkey script used a unique seed as input for each event sequence to prevent sequence repetition. The SARSA and Monkey experiments are run ten times on each application, with a two-hour time limit for each run. Test case termination criteria are probabilistic with a probability value of 0.05. The test case terminates by clicking on the Home button. A two-second delay was used after each event to ensure that the next event is not executed before the AUT responds. Two machines with identical configurations, Ubuntu 20.04.3 LTS with 32 GB RAM and a Pixel_3a emulator with API 29 (Android 10.0), are used to generate the test cases.

Table 3.6. Input Parameters for both SARSA and Monkey

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

A large starting Q-value of 500 was set for SARSA; this value will always be larger

than the values the agent derives after interacting with the environment, and the agent will select every event in a given state at least one time. Table 3.6 shows test generation parameters. After executing ten runs for each application, code coverage was collected and the average was reported.

TABLE 3.7. Average code coverage achieved by SARSA and Monkey testsuites

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

3.4.6. Results and Discussion

Table 3.7 represents the average line, branch, method, and class coverage achieved by SARSA and Monkey across ten runs for the seven subject applications. It also shows the code coverage improvement by SARSA over Monkey.

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

SARSA outperforms Monkey in all subject applications in terms of line, branch, method, and class coverage. Line coverage improvement ranges from 9.87% to 24.79%. The t-test is employed to assess whether a statistically significant difference exists in code coverage between the SARSA-based and Monkey test generation techniques. The results of the t-test indicate a significant difference in line ($p < 0.0001$), branch ($p < 0.0001$), method ($p < 0.0001$), and class ($p < 0.0001$) coverage between both techniques.

AnkiDroid, which is the largest among the subject applications in terms of lines of code, achieved the highest line coverage improvement. It also achieves the highest method and class coverage improvement. The method and class coverage improvements range from 7.88% to 28.48%, and 3.74% to 35.02%, respectively. *SimpleReminder*, achieves the lowest line, method, and class coverage improvement. It has 1126 lines of code, making it the second smallest of the subject applications. After a close inspection of the source code of *SimpleReminder*, it was found that the uncovered lines of code are mostly related to

some services that trigger based on context changes such as system broadcast of time. The proposed technique only works on GUI events, and it does not support context and system events yet. Monkey can generate several system-level events, despite that, the proposed approach achieves higher coverage than Monkey.

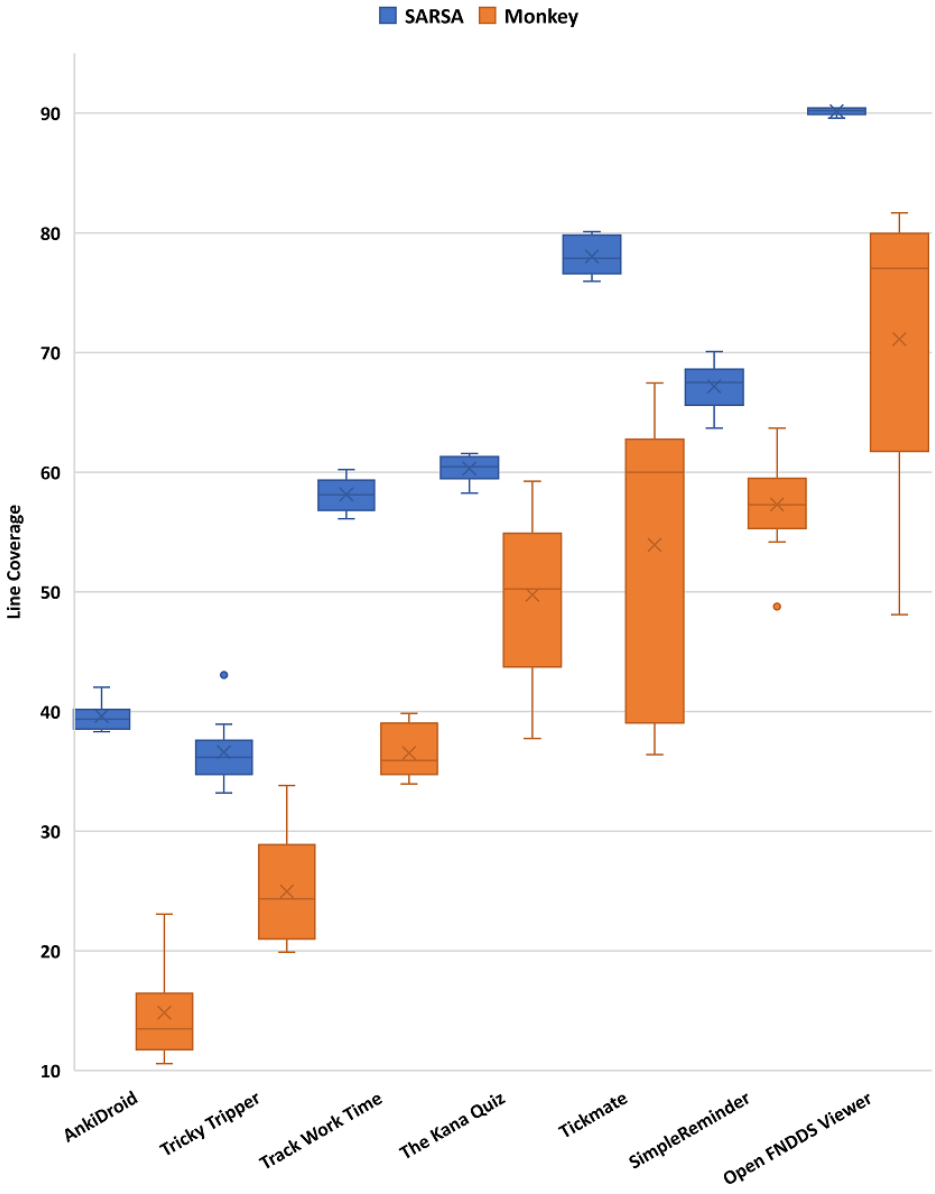


FIGURE 3.6. Line Coverage across all the subject applications and for all runs

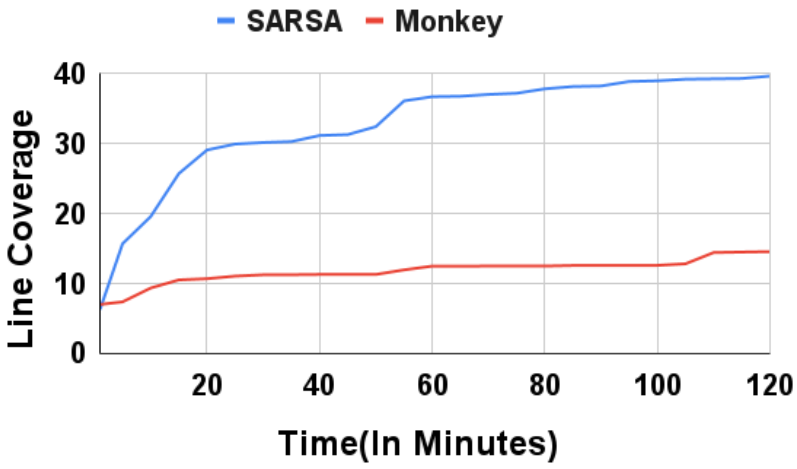
The highest branch coverage improvement 20.09 is achieved by *Tickmate*, while *The Kana Quiz* achieves the lowest branch coverage improvement. Branch coverage improvement ranges from 6.9% to 20.09%. Table 3.5 shows that *The Kana Quiz* has a relatively high

number of branches considering its total number of lines of code. It is a quiz application, and the quiz questions cover most of the code. The questions populate in the same activity. Since most of the possible events of this application are around the same activity and the app has high lines of code-to-branch ratio, Monkey can achieve a relatively high branch coverage. Therefore the branch coverage improvement by SARSA is lowest for this application. *Open FNDDS Viewer* is the smallest app with only 971 lines of code, and it achieved 100% class coverage by SARSA.

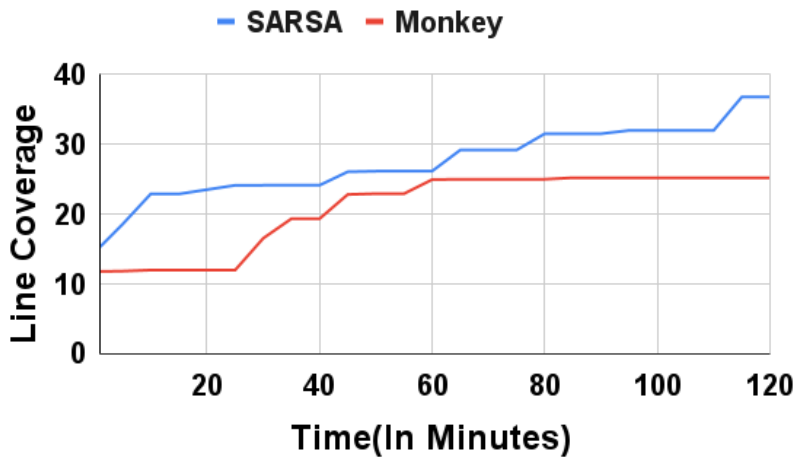
Figure 3.6 shows the distribution of line coverage achieved by both SARSA and Monkey in boxplots for all ten runs across all the subject applications. Branch, Method, and Class coverage follow similar patterns. SARSA achieves higher minimum, median, and maximum coverage than Monkey consistently for all the subject applications. In all the applications, Monkey achieves the minimum coverage value. SARSA boxplots are comparatively shorter than Monkey for the same application in all the cases, implying that the code coverage achieved by SARSA for an application across ten runs has less variation than Monkey.

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

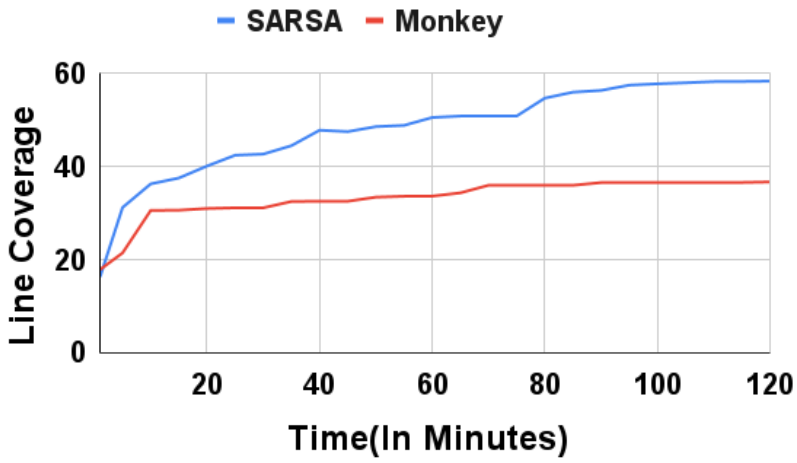
To evaluate the performance of both SARSA and Monkey over time, line coverage progress over time was observed for each subject application. The run among the ten runs that achieves the line coverage closest to the average value for each application is graphed, and the line coverage graph is plotted in Figure 3.7. Other runs show a similar pattern. The test generation time in minutes is plotted on the horizontal axis, and the vertical axis represents the achieved line coverage. An early jump is observed in small-sized applications, i.e., *Open FNDDS Viewer* and *Simple Reminder*. They reach the maximum or almost close to the maximum line coverage value by SARSA within 20 minutes. Similar patterns are observed in *Tickmate* and *The Kana Quiz*. *AnkiDroid*, *Tricky Tripper*, and *Track Work Time* are the larger applications among the test subjects in terms of lines of code. With these applications, SARSA outperforms Monkey from the beginning, but there is no early jump, and the progress is more pronounced.



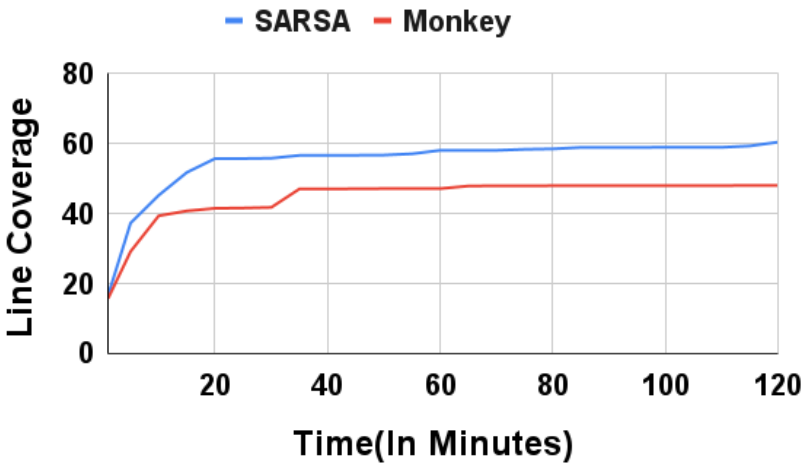
(A) AnkiDroid: Code coverage progress over time



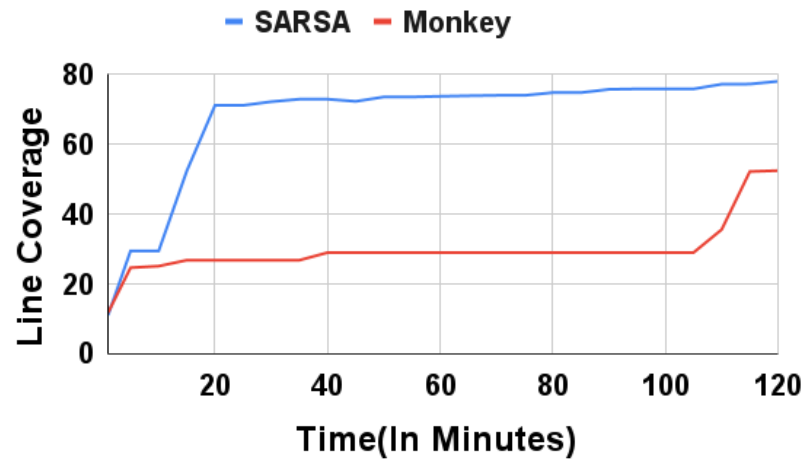
(B) Tricky Tripper: Code coverage progress over time



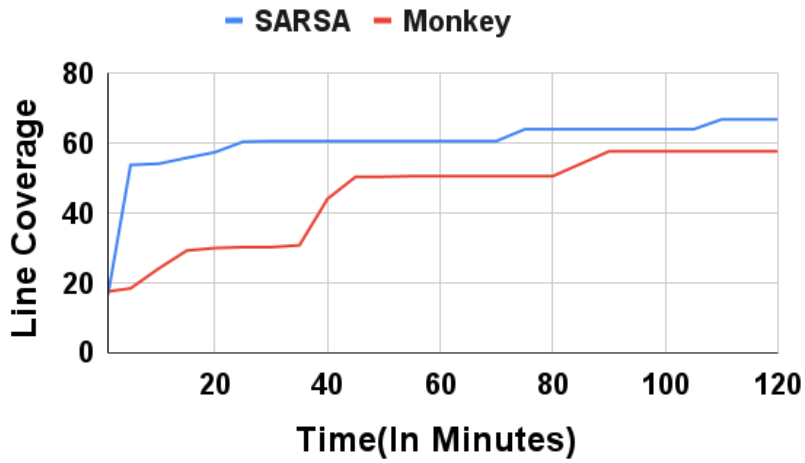
(C) Track Work Time: Code coverage progress over time



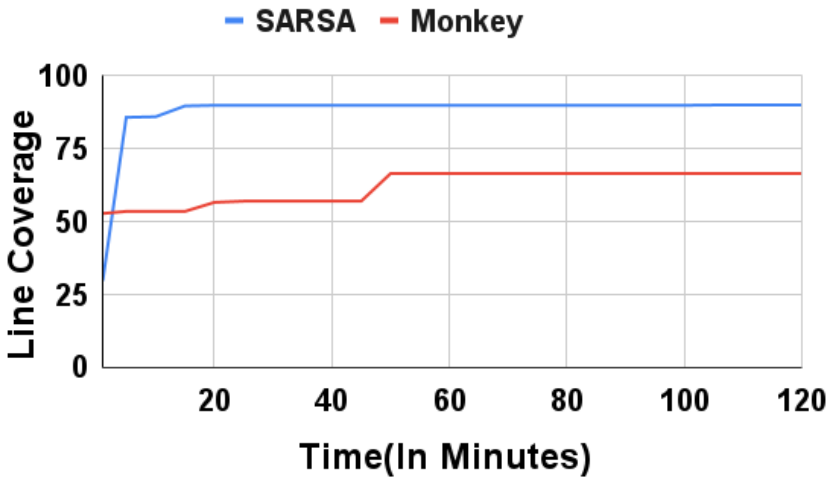
(D) The Kana Quiz: Code coverage progress over time



(E) Tickmate: Code coverage progress over time



(F) SimpleReminder: Code coverage progress over time



(G) Open FNDDS Viewer: Code coverage progress over time

FIGURE 3.7. Code coverage progress over time

After reaching a certain point, Monkey does not explore the AUT much but SARSA keeps exploring, given the strategy and parameters that encourage exploration. For all applications, SARSA-generated test suites achieve a faster rate of line coverage.

3.4.7. Threats to Validity

The validity of the study is subject to various threats.

Internal Validity: The internal validity of the study is subject to the choice of parameters for the SARSA algorithm. The *epsilon* value is critical in balancing exploration and exploitation throughout the learning process, and its choice can have a significant influence on the algorithm's behavior and results. In order to mitigate this concern, the study employed an epsilon value of 0.3 as a means of achieving a harmonious equilibrium between exploration and exploitation. To enhance the internal validity, future studies ought to consider experiments with different *epsilon* values or decayed *epsilon* to gain a better understanding of the trade-offs involved in this area. Furthermore, it is worth considering the exploration of various configurations concerning the learning rate and discount factor in prospective research endeavors.

External Validity: The SARSA-based test generation approach was evaluated in this study

utilizing seven Android applications. The limited number of applications raises questions regarding the generalizability of the results, as distinct applications, have unique characteristics that may result in variable outcomes. To limit this risk, attempts were taken to improve external validity by selecting applications of diverse sizes and classifications. By adding this variety, the research aimed to capture a greater sample of the Android application environment and boost the possible generalizability of the results.

Construct Validity: The test case termination criteria employed in the study are characterized by a probabilistic nature, which gives rise to concerns pertaining to the potential threat to validity resulting from the variability inherent in pressing the Home button. In order to mitigate this issue, a total of ten trials were executed for every application, and the resulting averages were compiled.

CHAPTER 4

PAIRWISE INTERACTION, ACTIVITY, AND APPLICATION STATE COVERAGE-BASED PRIORITIZATION TECHNIQUES TO IMPROVE CODE COVERAGE FOR SARSA GENERATED TEST CASES

Android applications featuring graphical user interfaces (GUIs) are built around activities and are driven by events. Individual screens of Android applications are called activities. An activity is similar to a page on a website. Each activity encompasses a variety of GUI components and widgets, notably including buttons, text boxes, and images, that allow for user engagement and interaction. An application may comprise several activities, and end-users traverse through them to access various features and functionalities.

The events triggered through user interactions with the GUI of an application characterize its functional behavior. Let's consider the following example to better understand the concept of an event: Assume we have a messaging app that has a "Send" button. Clicking on the "Send" button triggers an event. This event indicates the user's desire to send a message. The application process the event, which includes activities such as verifying the message content, delivering the message to the recipient, and updating the message history. The event-driven architecture of the applications guarantees that the relevant steps are completed when the "Send" button is tapped, allowing for smooth communication between users.

The event interactions by the users cause the application to transition between different states based on its GUI, internal logic, and functionality. For instance, clicking a button may lead to a new screen being displayed or a specific action being performed. The confluence of GUI events and their resultant application states culminates in a dynamically evolving interaction model. Inspired by the work of Bryce et al. [27] [29], this study proposes new prioritization strategies for Android applications that take into account the application's state and activity, as well as pairwise event interaction coverage. The prioritization techniques proposed in this study aimed to attain comprehensive coverage of distinctive pairs of

events, activities, and states within the Android application context. The primary objective of this study is to improve the code coverage rate of test cases generated through the SARSA method by prioritizing test cases that cover a diverse range of event pairs, activities, and states.

4.1. Proposed Prioritization Strageies

The study adopts the definitions of states and events employed in test case generation, which are outlined in detail in Chapter 3.

A state is defined as an n-tuple: $s = (a_1, a_2, a_3, ..., a_n)$ where every a_i is a legal action available to be enacted on the on-screen elements.

Events are a set of one or more related actions that may occur in a GUI state. An event and its ordered set of actions are represented in a 2-tuple $e = (s, A_e)$. In this representation, the GUI state is s and the ordered set of actions associated with the event is A_e. An example event could be a single click on the login button of an application or an input of several sentences into a text field.

This work introduces four strategies for prioritizing test suites, which encompass pairwise coverage, state coverage, and activity coverage. These strategies are as follows:

(1) Pairwise event interaction coverage,

(2) Pair-Activity coverage (PA),

(3) Pair-State coverage(PS), and

(4) Pair-State-Activity coverage (PSA).

The first strategy focuses solely on covering unique event pairs, while the other three strategies incorporate activity and state interactions alongside pair coverage to compute prioritization scores.

4.1.1. Pairwise event interaction coverage based prioritization

The pairwise event interaction coverage-based prioritization algorithm reorders test cases using the number of unique event pairs that each test covers. To determine the prioritization score for each test case, the algorithm generates pairs of events and then calculates

the score using the following equation:

$$pScore = pCount \tag{4.1}$$

where $pScore$ denotes the prioritization score assigned to the test case, while $pCount$ represents the number of pairs covered by that particular test case.

This strategy measures the degree of coverage supplied by each test case in terms of event interactions by setting the priority score equal to the count of unique event pairs covered. This approach serves to optimize the coverage of critical code while mitigating limitations in resource allocation. The algorithm places emphasis on the coverage of unique event pairs as a means of ensuring that selected test cases possess the capacity to furnish extensive and inclusive coverage of event interactions. This helps to identify the test cases with intricate functionalities and complex system behaviors. Assigning high prioritization scores to these critical test cases increases the chances of detecting critical defects and revealing latent issues that may emerge due to particular event combinations or inter-dependencies.

4.1.2. Pair-Activity coverage based prioritization(PA)

The Pair-Activity coverage-based prioritization (PA) algorithm presented in this work offers a comprehensive approach to test case prioritization by considering both the number of event pairs covered and the number of unique activities within each test case. By incorporating these two factors, the algorithm aims to prioritize test cases that provide balanced coverage of event interactions and activities, leading to a more thorough testing process.

To calculate the prioritization score for each test case, the PA algorithm follows a multi-step process. First, it generates all unique event pairs within the test case that are not already covered by previously selected test cases, representing the potential interactions between events. Next, it counts the number of unique activities present in the test case.

$$pScore = pCount * cActivity \tag{4.2}$$

In this equation, $pScore$ represents the prioritization score assigned to the test case,

pCount signifies the number of covered event pairs, and *cActivity* denotes the number of unique activities within the test case. By multiplying the count of event pairs by the number of unique activities, the algorithm captures the combined coverage of both event interactions and activities. By prioritizing test cases based on Pair-Activity coverage, the algorithm enhances the effectiveness of the testing process. It enables the identification of test cases that not only cover a wide range of event interactions but also exercise distinct activities within the system. This comprehensive coverage helps in uncovering unexplored activities thereby maximizing coverage rate as well as fault detection probability.

4.1.3. Pair-State coverage based prioritization(PS)

The Pair-State coverage-based prioritization (PS) algorithm gives priority to test cases based on the number of event pairs covered by a test case and the number of states present within a test case. This strategy aims to prioritize test cases that provide a thorough exploration of functional behaviors and state transitions. It generates all unique event pairs and counts the number of unique states before calculating the prioritization score for each test case using the following equation:

$$pScore = pCount * cState \tag{4.3}$$

In this equation, *pScore* is the prioritization score, *pCount* is the number of event pairs covered by the test case, and *cState* is the number of unique states.

By multiplying the count of event pairs by the number of unique states, the algorithm effectively captures the combined coverage of both event interactions and application states. By considering these two factors, the algorithm ensures that test cases not only exercise various event interactions but also cover application state transitions. The goal is to promote a more realistic and thorough testing approach that facilitates the identification of test cases that not only cover a diverse range of event interactions but also explore different system states.

4.1.4. Pair-State-Activity coverage based prioritization(PSA)

The Pair-State-Activity coverage-based prioritization (PSA) algorithm offers a sophisticated approach to prioritizing test cases by considering the coverage of event pairs, activities, and states. The PSA strategy aims to prioritize test cases that provide a comprehensive exploration of system behaviors, activities, and state transitions by incorporating the aforementioned three factors. To calculate the prioritization score, it generates all unique event pairs and counts the pairs that are not covered by previously selected test cases, and counts the number of unique activities and states present in each test case. It then calculates the prioritization score for each test case using the following equation:

$$pScore = pCount * cState * cActivity \tag{4.4}$$

In this equation, *pScore* denotes the prioritization score assigned to the test case, *pCount* represents the number of event pairs covered by the test case, *cActivity* signifies the count of unique activities, and *cState* indicates the count of unique states within the test case.

The PSA algorithm captures the combined coverage of event interactions, activities, and states by multiplying the count of event pairs by the number of unique activities and unique states. This promotes a comprehensive and realistic testing approach and enhances the effectiveness and efficiency of the testing process. This comprehensive coverage contributes to a broader coverage of code segments associated with specific states or activities. When test cases cover a wide range of event pairs, activities, and states, they inherently trigger different execution paths in the code. As a result, the code segments associated with these paths are more likely to be executed, leading to increased code coverage. Prioritizing test cases with high coverage results in a higher coverage rate and the likelihood of finding defects early.

4.2. Algorithm Overview

The pseudocode for the test case prioritization process is illustrated in Algorithm 4. It takes a test suite as input and produces a prioritized test suite as output.

Algorithm 4: Pair-State-Activity coverage based prioritization

Input: Test Suite T
Output: Prioritized test suite T'

```
1  T' = []
2  while uncovered event pairs remain do
3      current_max = −1
4      current_best = NULL
5      for test case t_i in T do
6          if t_i not in T' then
7              compute pCount as number of unique event pairs covered by t_i
8              count unique activities covered by t_i
9              count unique states covered by t_i
10             compute pScore as prioritization score
11             if pScore > current_max then
12                 current_best = t_i
13                 current_max = pScore
14             end
15             else if pScore == current_max then
16                 Tie is broken at random
17             end
18         end
19     end
20     add current_best to T'
21     remove current_best from T
22     Mark all unique pairs covered by current_best as covered
23 end
24 Add the remaining tests in T in random order to T' return T'
```

Given an input test suite T, the algorithm generates a prioritized test suite T' by iteratively selecting the test case with the highest prioritization score and adding it to an ordered list. The algorithm first initializes an empty list T' to store the prioritized test suite. It then enters a while loop that iterates until all event pairs are covered. For each test case t_i in the original test suite T, the algorithm counts unique event pairs that are not covered by the already selected test cases, unique activities, and unique states. Based on the selected prioritization strategy, the algorithm computes the prioritization score. The pseudocode shown in the algorithm is for the PSA strategy. Other strategies follow similar steps. The algorithm marks the test case with the highest score as *current_best*. If two or more test case has the same score, the algorithm breaks the tie randomly. After checking all the test

cases, the algorithm adds the best test case to the prioritized suite. Once all event pairs are covered, the algorithm exits the while loop. It then adds the remaining test cases from the original test suite T (which were not selected during the prioritization process) to T' in random order.

4.3. Empirical Analysis

4.3.1. Research Questions

The experiments examine the following research questions:

RQ1. **Does prioritization by pairwise event interaction coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?**

RQ2. **Does prioritization by pair-activity coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?**

RQ3. **Does prioritization by pair-state coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?**

RQ4. **Does pair-state-activity coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?**

TABLE 4.1. Characteristics of subject applications

App Name	# LOC	# Branches	# Methods	# Classes	#Activity
AnkiDroid	29063	11772	4091	500	21
Tricky Tripper	8244	2512	1766	290	16
The Kana Quiz	4453	2231	629	87	7
Tickmate	2654	770	395	60	10
SimpleReminder	1126	314	292	48	4

4.3.2. Subject Applications

To evaluate the effectiveness of the proposed test suite prioritization techniques, this study employed test suites from five different Android applications:

(1) AnkiDroid

(2) The Kana Quiz

(3) SimpleReminder

(4) Tickmate and

(5) Trickytripper.

These open-source applications are obtained from F-droid, an online repository of open-source Android applications [5]. Table 4.1 shows the characteristics of the five subject applications. These characteristics include the lines of code (LOC), branches, methods, classes, and activities for each application. The range of applications varies significantly in terms of their coding composition, with code lengths ranging from 1126 to 29063 lines, branches ranging from 314 to 11772, methods varying from 292 to 4091, classes varying from 48 to 500, and activities ranging from 4 to 21. *AnkiDroid* is the largest among the five applications for all of the above metrics. *The Kana Quiz* and *Tricky Tripper* are also relatively large, while *Tickmate* and *SimpleReminder* are smaller in size. The number of activities also varies among the applications, with *AnkiDroid* having the most and *SimpleReminder* having the least.

TABLE 4.2. Characteristics of test suites

App Name	**# testcases**	**# Line Coverage**	**# Branch Coverage**	**# Activity Covered**
AnkiDroid	120-148	39.62	25.16	14-16
Tricky Tripper	106-125	36.61	20.27	8-10
The Kana Quiz	101-124	63.3	46.71	6
Tickmate	94-116	78.02	57.87	10
SimpleReminder	106-125	67.18	48.89	4

4.3.3. Test Generation

SARSA, a reinforcement learning technique, generates test suites for the subject applications during the testing process [61] as described in Chapter 3. SARSA uses a trial-and-error approach to identify the best GUI events that will maximize code coverage. SARSA-based test generation was run for two hours to generate one test suite per application, and ten test suites were developed for each application. SARSA creates test cases by navigating the GUI without requiring access to the application's source code. However, in the study, the source code was utilized to gather code coverage data using JaCoCo [8]. The only requirement for generating test cases with SARSA is the Android Package file instrumented

with JaCoCo. The test suites utilized in this study contained around 94-148 test cases. The proposed test suite prioritization algorithms were then tested on these test suites to evaluate their effectiveness. Table 4.2 shows the characteristics of the test suites generated by SARSA, including the number of test cases in the test suite, average line and branch coverage, and the range of activities covered by the generated test suites.

4.3.4. APSC and APBC Metrics

A common metric for evaluating test case prioritization algorithms and their effectiveness is APFD (Average Percentage of Faults Detected), which calculates the rate of fault coverage by the prioritization algorithm [86]. Variations of this metric, such as APFD_C, also consider fault severity and test case cost [87].

This research focuses on maximizing the code coverage rate; therefore, it utilizes the APSC (Average Percentage of Statement Coverage) and APBC (Average Percentage of Branch Coverage) metrics introduced by Li et al. [74] to evaluate the performance of the proposed test case prioritization algorithms. These metrics are inspired by APFD and are calculated using the following formulas:

$$APSC = 1 - \frac{TS_1 + TS_2 + ... + TS_m}{nm} + \frac{1}{2n}$$

$$APBC = 1 - \frac{TB_1 + TB_2 + ... + TB_m}{nm} + \frac{1}{2n}$$

where TS_i and TB_i represent the first test case that covers the statement or branch i, the variable m denotes the total count of statements that the test suite covers and n represents the total number of tests in the test suite.

These metrics provide valuable information about code coverage, and a higher percentage generally indicates that the algorithm is more effective at maximizing code coverage quickly. The study expects the APSC and APBC scores of a prioritized test suite will increase compared to the default ordering, as the anticipation is that more statements and branches will be covered earlier in the testing process.

4.3.5. Experimental Setup

A comprehensive assessment is conducted to evaluate the efficacy and potency of the proposed strategies for prioritizing SARSA-generated test cases. The test case prioritizing algorithms are applied systematically to the test suites of the five subject applications, with default and random orderings serving as comparative baselines. The test cases in each test suite are reordered based on each prioritization technique using Algorithm 4, implemented through Python scripts. To ensure the dependability of the outcomes, every prioritization methodology is executed on every test suite for a cumulative count of ten iterations. The utilization of a multiple-run approach facilitates the consideration of conceivable variations and affords a more precise depiction of the efficacy of each respective technique. Coverage reports are gathered to undergo analysis for every prioritized test suite. Moreover, the APSC and APBC scores are calculated, and the mean scores for every application are ascertained, yielding valuable insights pertaining to their impact on coverage.

4.4. Results and Discussion

Tables 4.3 and 4.4 present the average APSC and APBC scores for the prioritization strategies applied to the five subject applications. The study compares the scores of the prioritization strategies to the test suites' original order and randomized orderings. The tables 4.5 and 4.6 show the differences in APSC and APBC scores between prioritization strategies and the default and random orderings. These tables provide insight into the changes in scores that occurred after prioritization. On average, the pairwise, PA, PS, and PSA coverage-based prioritization strategies all outperform the default and random ordering strategies in terms of APSC and APBC scores for all subject applications.

Table 4.3. APSC Results by applications

Technique	**AnkiDroid**	**Trickytripper**	**The Kana Quiz**	**Tickmate**	**SimpleReminder**
Default	92.78%	93.03%	94.83%	88.86%	92.92%
Random	94.11%	94.59%	94.78%	90.01%	93.62%
Pairwise	98.21%	99.07%	97.68%	96.84%	97.86%
PA	98.32%	99.27%	98.08%	97.01%	98.07%
PS	98.25%	99.13%	97.41%	96.52%	97.99%
PSA	98.32%	99.26%	97.76%	96.73%	98.09%

TABLE 4.4. APBC Results by applications

Technique	AnkiDroid	Trickytripper	The Kana Quiz	Tickmate	SimpleReminder
Default	94.91%	95.53%	94.76%	90.19%	92.09%
Random	95.87%	96.58%	95.10%	90.82%	92.80%
Pairwise	98.86%	99.55%	98.52%	97.32%	98.13%
PA	98.92%	99.66%	98.70%	97.51%	98.30%
PS	98.90%	99.58%	98.31%	97.08%	98.27%
PSA	98.93%	99.65%	98.51%	97.28%	98.35%

RQ1: Does prioritization by pairwise event interaction coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?

The pairwise event interaction coverage provides higher scores for APSC and APBC compared to using default or random ordering methods for all five subject applications. The pairwise coverage method was found to have an average improvement of 2.84-7.98% in APSC and 3.76-7.13% in APBC when compared to the default order. Furthermore, when compared to random ordering, the pairwise coverage method had an average improvement of 2.90-6.83% in APSC and 2.98-6.50% in APBC. The results for all applications using the Pairwise strategy are as follows: *AnkiDroid* surpassed the default order in APSC and APBC by 5.43% and 3.95%, respectively. In comparison to the random order, the findings show a 4.10% and 2.99% improvement in APSC. *Trickytripper* outperformed the default order in both APSC and APBC, yielding improvements of 6.03% and 4.03%, respectively. In comparison to the random order, the acquired findings showed a 4.48% increase in APSC and a 2.98% increase in APBC. *The Kana Quiz* showed a 2.84% improvement in APSC and a 3.76% improvement in APBC when compared to the default ordering. *The Kana Quiz* showed a 2.90% improvement in APSC and a 3.43% improvement in APBC when compared to the random sequence.

Tickmate attained a superior APSC of 7.98% and an enhanced APBC of 7.13% when compared to the default order. In contrast to the random order, the *Tickmate* exhibited a significant improvement of 6.83% and 6.50% in APSC and APBC, respectively. *SimpleReminder* was observed to yield a 4.93% improvement in APSC and a 6.04% enhancement

in APBC when compared to the default setting. In contrast to the random order, APSC improved by 4.24% and APBC improved by 5.33%. The present findings suggest that the adoption of the Pairwise approach resulted in a notable enhancement in both APSC and APBC when compared to the default and random configurations across all applications.

RQ2: Does prioritization by pair-activity coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?

The results show that pair-activity coverage-based prioritization increases APSC over default and random ordering techniques for all five subject applications. Pair-activity coverage-based prioritization technique shows an average increase of 3.24-8.14% in APSC and 3.94-7.32% in APBC when compared to the default ordering method. Additionally, when compared to the random ordering method, the pair-activity coverage-based prioritization technique shows an average increase of 3.30-7.00% in APSC and 3.04-6.69% in APBC. For *AnkiDroid*, the PA strategy achieved 5.53% better APSC and 4.00% better APBC compared to the default order. Compared to random order, it demonstrated 4.20% better APSC and 3.04% better APBC. The results indicate that the PA algorithm demonstrated a significant improvement of 6.24% in APSC and 4.14% in APBC when compared to the default setting for *Trickytripper*. In comparison with the random order, the *Trickytripper* results displayed a 4.68% improvement in APSC and a 3.09% enhancement in APBC. *The Kana Quiz* demonstrated an improvement of 3.24% in APSC and 3.94% in APBC when compared to the default order. In contrast to the random order, the data exhibited a 3.30% improvement in APSC and a 3.60% enhancement in APBC. *Tickmate* demonstrated a significant improvement of 8.14% in APSC and 7.32% in APBC, as compared to the default order. In comparison to the random order, the results indicated a superior APSC by 7.00% and an improved APBC by 6.69%. The study found that the utilization of the PA strategy on *SimpleReminder* resulted in a 5.15% improvement in APSC and a 6.21% enhancement in APBC relative to the default order. In comparison to the random order, the results indicate a 4.45% enhancement in APSC and a 5.50% improvement in APBC.

TABLE 4.5. Δ APSC by applications

	AnkiDroid		**Trickytripper**		**The Kana Quiz**		**Tickmate**		**SimpleReminder**	
Technique	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom
Pairwise	5.43%	4.10%	6.03%	4.48%	2.84%	2.90%	7.98%	6.83%	4.93%	4.24%
PA	5.53%	4.20%	**6.24%**	**4.68%**	**3.24%**	**3.30%**	**8.14%**	**7.00%**	5.15%	4.45%
PS	5.47%	4.14%	6.10%	4.54%	2.58%	2.63%	7.65%	6.51%	5.07%	4.37%
PSA	**5.54%**	**4.21%**	6.23%	4.67%	2.92%	2.97%	7.86%	6.72%	**5.17%**	**4.47%**

TABLE 4.6. Δ APBC by applications

	AnkiDroid		**Trickytripper**		**The Kana Quiz**		**Tickmate**		**SimpleReminder**	
Technique	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom
Pairwise	3.95%	2.99%	4.03%	2.98%	3.76%	3.43%	7.13%	6.50%	6.04%	5.33%
PA	4.00%	3.04%	**4.14%**	**3.09%**	**3.94%**	**3.60%**	**7.32%**	**6.69%**	6.21%	5.50%
PS	3.98%	3.02%	4.06%	3.00%	3.55%	3.21%	6.90%	6.27%	6.18%	5.47%
PSA	**4.01%**	**3.05%**	4.13%	3.08%	3.75%	3.41%	7.09%	6.46%	**6.26%**	**5.55%**

RQ3: Does prioritization by pair-state coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?

The pair-state coverage-based prioritization methods demonstrate better APSC and APBC scores than the default and random ordering techniques for all five target applications. To be precise, the pair-state coverage-based prioritization technique has an average increase of 2.58-7.65% in APSC and 3.55-6.90% in APBC when compared to the default ordering method. When comparing the pair-state coverage-based prioritization technique to the random ordering method, it was found that there was an average increase of 2.63-6.51% in APSC and 3.00-6.27% in APBC.

For *AnkiDroid*, the PS strategy outperformed the default order with 5.47% better APSC and 3.98% better APBC. It also achieved 4.14% better APSC and 3.02% better APBC compared to random order. *Trickytripper* achieved 6.10% and 4.54% better APSC compared to default and random order respectively. It also demonstrated superior APBC scores compared to the default and random order, with improvements of 4.06% and 3.00% respectively. *The Kana Quiz* showed APSC improvements compared to the default and random order, with increases of 2.58% and 2.63% respectively. Additionally, the PS strategy achieved 3.55% better APBC compared to the default and a 3.21% better APBC over random order for *The Kana Quiz*. The APSC enhancement observed for *Tickmate* amounted to 7.65% in contrast to default, and 6.51% when compared to random order. For the same

application, PS demonstrated a superiority of 6.90% in APBC compared to default, and a notable advantage of 6.27% in APBC when compared to a random order. The utilization of PS on *SimpleReminder* yielded improvement by a 5.07% increase in APSC and a 6.18% increase in APBC when contrasted with the default ordering. When compared with the random order, the PS strategy demonstrated a 4.37% increase in APSC and a 5.47% increase in APBC.

RQ4: Does pair-state-activity coverage-based strategy achieve higher APSC and APBC scores than default and random ordering?

The results of the study indicate that the pair-state-activity coverage-based prioritization strategy is a highly effective method for increasing code coverage in Android applications. This method was found to be superior to both the default and random ordering techniques in terms of APSC and scores for all five subject applications examined. The utilization of the pair-state-activity coverage-based prioritization technique resulted in an average improvement of 2.92%-7.86% in APSC and 3.75%-7.09% in APBC in comparison to the default ordering. Moreover, an average upsurge of 2.97%-6.72% in APSC and 3.05%-6.46% in APBC was observed when compared to random ordering.

The *AnkiDroid* exhibits a 5.54% enhancement in APSC in contrast to the default ordering, along with a 4.21% improvement compared to the random strategy. Similarly, it exhibited a 4.01% enhancement in APBC as compared to the default method, and a 3.05% improvement when compared to the random ordering strategy. For *Trickytripper* PSA showed a 6.23% rise when compared to the default order and a 4.67% growth in comparison to a random strategy. It also demonstrated a 4.13% increase compared to the default and a 3.08% boost in comparison to a random order in APBC. *The Kana Quiz* application achieved a 2.92% improvement in APSC with PSA over default and a 2.97% improvement over random. Additionally, it achieved a 3.75% increase in APBC over default and a 3.41% increase over random prioritization. The PSA approach boosted APSC by 7.86% over the default and by 6.72% over random order in *Tickmate*. Similarly, it increased APBC by 7.09% compared to the default and by 6.46% compared to random order. PSA amplified

SimpleReminder's APSC by 5.17% and 4.47%, respectively, outperforming the default and random order. Furthermore, it resulted in a 6.26% gain in APBC over the default and a 5.55% increase over random order prioritization.

Discussion: A statistical analysis using the t-test was conducted to examine the significance of the differences in APSC and APBC scores achieved by the proposed strategies(pairwise, PA, PS and PSA) compared to default and random strategies. The t-test results revealed a highly significant difference ($p < 0.001$) in APSC and APBC scores for all the proposed techniques when compared to both the default and randomly ordered test suites.

The default and random ordering strategies consistently achieved the lowest scores for all subject applications, while the pairwise, PA, PS, and PSA strategies consistently achieved higher scores. Among these strategies, the PSA strategy performed particularly well for the *AnkiDroid* and *SimpleReminder* applications, while the PA strategy performed the best for the *Trickytripper*, *The Kana Quiz*, and *Tickmate* applications. Tables 4.5 and 4.6 highlight the top-performing strategies for each application. Although the difference is marginal, it appears that the PA and PSA techniques, which incorporate activity coverage, consistently achieve the highest APSC and APBC scores across all applications.

The improvement in scores is lowest for *The Kana Quiz* application and highest for *Tickmate*. This may be due to the fact that *The Kana Quiz* is a quiz app with most of its functionality concentrated in one activity, while *Tickmate* test suites achieve the highest code coverage among the applications. Further, the test cases cover all 10 of its activities. *AnkiDroid* has the highest LOC among the applications under test, and its test suites only covered 14-16 activities out of 21, which may not have been sufficient to thoroughly test the application. *Trickytripper* may also have suffered from inadequate test coverage. The PS strategy, which utilizes state coverage, did not achieve the highest score for any of the applications. Generally, it achieved marginally higher scores than Pairwise and slightly lower than the PA and PSA strategies, which utilize activity coverage. This suggests that state coverage alone may have a lower level of impact on APSC and APBC scores compared to

activity coverage. The PSA strategy, which considers both activity and state coverage, tends to achieve higher APSC and APBC scores for some applications compared to the strategies that only consider one type of coverage. This suggests that both activity coverage and state coverage are important factors in achieving high APSC and APBC scores.

4.5. Threats to Validity

Internal Validity: The proposed strategies in this study showed improvements over the default and random ordering methods. However, one potential limitation is the number of test cases per test suite, which may impact the accuracy of results, especially for larger applications like *Ankidroid* and *Trickytripper*. Another consideration is the influence of randomness on the algorithms, as the APSC/APBC scores calculated using random ordering can exhibit significant variance. To address this, the study conducted multiple runs of each algorithm on each test suite and reported the average results, including random ordering. Additionally, the use of randomness to break ties and the ordering of final test cases were consistent across all proposed strategies.

External Validity: The generalizability of the study's findings may be limited due to the small number of applications tested. Despite efforts to choose applications of varying magnitudes, a bigger and more diverse sample of applications would offer a more complete picture of the algorithms' efficacy. It should be noted that the study focused solely on test suite priority for Android applications, and the results may not be relevant to other types of applications.

Construct Validity: The study focused on assessing the APSC and APBC scores as measures of effectiveness for the proposed strategies. The experimental design included variations in test suite ordering and the use of different prioritization algorithms. However, there may be additional factors or metrics that could impact the construct validity of the study. Future research could explore other measures or consider different aspects of test suite prioritization to further validate the findings.

Conclusion Validity: In conclusion, the study suggests that the proposed strategies for test suite prioritization in Android applications can lead to improved APSC and APBC scores

compared to default and random ordering methods. However, the constraints indicated, such as the number of test cases per test suite and the restricted number of apps examined, should be noted when interpreting the results. Further study with a bigger and more diverse sampling of applications is required to improve the findings' validity and generalizability.

CHAPTER 5

ELEMENT COVERAGE AND WEIGHTED COST-BASED PRIORITIZATION TECHNIQUES FOR ANDROID TEST SUITES GENERATED BY A REINFORCEMENT LEARNING ALGORITHM

Chapter 4 of this dissertation has focused on the prioritization of test cases generated by the SARSA algorithm and it has introduced novel methodologies that combine pairwise event interaction coverage, application state coverage, and activity coverage. The prioritization based on activity coverage has exhibited promising potential in enhancing the APSC and APBC scores across multiple tested applications. This finding suggests a correlation between UI coverage and code coverage rate. Building upon these insights, the current research endeavors to investigate the impact of UI element coverage on optimizing test case execution to achieve higher code coverage rates.

To this end, this chapter introduces Android test case prioritization strategies that incorporate unique UI element coverage alongside considerations of test case cost and complexity. A key concept introduced is the notion of "test case weight," which allows for the weighting of test case complexity. Through the integration of SARSA-based test case generation with multifactor test case prioritization, the present study aims to enhance the efficacy of test case execution, leading to an eventual improvement in code coverage rates for Android applications.

5.1. Prioritization Strategies and the algorithm overview

This section outlines the proposed strategies, provides the definition of "test case weight" and presents the test case prioritization algorithm.

5.1.1. Element Coverage-based Prioritization($\boldsymbol{ECP}$)

The Element Coverage-based Prioritization (ECP) method is a software testing approach that prioritizes test cases by analyzing the coverage of distinctive user interface (UI) elements that other test cases have not been previously covered. With the aim of enabling

the testing process to explore a broader range of areas of the application within a constrained period of time, this method seeks to find and execute critical test cases that cover an extensive number of different UI elements. In the context of an Android application, an element can be any of the various types of UI components, including but not limited to buttons, text boxes, and dropdown boxes.

This technique determines the number of unique UI elements in a test case that is not previously covered by any other test case to obtain the prioritization score for a specific test case. The prioritization score of a test case is positively related to its coverage of unique UI elements. The following equation computes the prioritization score of a test case:

***ECP* :**

$$p\_score = e\_count \tag{5.1}$$

Where *p_score* denotes the prioritization score, and *e_count* represents the distinct UI elements count in the test case which is not already covered by another test case in the prioritized test suite.

5.1.2. Element Coverage and Cost-based Prioritization($ECCP$)

The Element Coverage and Cost-based Prioritization ($ECCP$) strategy combines the coverage of unique UI elements with the cost of executing test cases to optimize resource usage and time. The goal of the $ECCP$ strategy is to identify and execute critical test cases early, which encompass a considerable proportion of unique UI elements while simultaneously striving to maintain low execution costs.

Two distinct variations of $ECCP$ are employed, namely, $ECCP_1$ and $ECCP_2$, each utilizing a different method for computing the prioritization score.

***ECCP*$_1$:**

$$p\_score = \frac{e\_count}{cost} \tag{5.2}$$

***ECCP*$_2$:**

$$p\_score = e\_count + \frac{1}{cost} \tag{5.3}$$

The main difference between the $ECCP_1$ and $ECCP_2$ equations is how they balance the importance of element coverage and the cost of the test case to calculate the prioritization score. In $ECCP_1$, the prioritization score is directly proportional to the element coverage and inversely proportional to the cost of the test case. This approach prioritizes test cases in accordance with the number of user interface (UI) elements covered per unit cost. On the other hand, $ECCP_2$ calculates the prioritization score as the sum of the number of unique UI elements covered by the test case and the inverse of the cost of the test case. The inverse of the cost is added to the element counts as a penalty term. This way, $ECCP_2$ gives more importance to the element coverage. When a test case encompasses a significant quantity of UI elements, the prioritization score will be elevated despite the test case's considerable cost. In summary, the $ECCP_1$ methodology places significant emphasis on prioritizing cost efficiency, while the $ECCP_2$ approach employs a nuanced perspective that factors in not only the scope of UI elements encompassed but also the related test case costs.

The measurement of the cost of test case execution can be assessed through various means including the duration of executing the test case, the resources necessary for its execution, and the intricacy associated with the test case. The specific method for measuring the cost may differ depending on the particular requirements of the testing process. The proposed strategies investigate the execution time and length of the test case as the cost of a test case, leading to four different ways of computing the prioritization score for $ECCP$:

***ECC*$_T$*P*$_1$:**

$$p\_score = \frac{e\_count}{execution\_time} \tag{5.4}$$

***ECC*$_L$*P*$_1$:**

$$p\_score = \frac{e\_count}{t\_length} \tag{5.5}$$

ECC$_T$P$_2$:

$$p\_score = e\_count + \frac{1}{execution\_time} \tag{5.6}$$

ECC$_L$P$_2$:

$$p\_score = e\_count + \frac{1}{t\_length} \tag{5.7}$$

In the above equations, the variable t_length denotes the total count of events in the test case, while $execution\_time$ indicates the duration of test case execution in seconds, which is recorded during test case generation.

5.1.3. Element Coverage and Weighted Cost-based Prioritization ($ECWCP$)

Element Coverage and Weighted Cost-based Prioritization ($ECWCP$) is an approach that builds upon the existing $ECCP$ technique. While $ECCP$ techniques combine the element coverage and cost of the test case for prioritization, they don't take into account the complexity of the test cases. $ECCP_1$ strategies assume that all the test cases in a test suite have similar complexity and prioritize them based on the ratio of element coverage and cost. $ECCP_2$ also considers all the test cases of equal complexity but unlike $ECCP_1$ it adds the inverse of cost as a penalty term to the element coverage to prioritize the test cases.

However, test case complexity can greatly influence the coverage rate, as certain actions may hold more importance than others. Therefore, the introduction of test case weight is proposed to address the complexity of the test cases during prioritization, resulting in the $ECWCP$ approach.

Test Case Weight: The notion of "test case weight" denotes a numerical value assigned to a test case based on its significance in exploring new areas within the application by the actions it encompasses.

Definition 5.8. The total weight of a test case T_i is defined as:
$W(T_i) = \sum w(a)$, for all a in $A(T_i)$,

where $W(T_i)$ denotes the total weight of test case T_i, $A(T_i)$ represents the set of actions contained in T_i and $w(a)$ refers to the weight assigned to action a.

TABLE 5.1. Weight Assignment for different action types

Action	**Weight**
click	3
long-click	3
text-entry	3
enter	3
run-in-background	3
check-uncheck	2
back	2
swipe-up	1
swipe-down	1
swipe-right	1
swipe-left	1
home	1
launch	1

In other words, the weight of a test case is the sum of the weights of all the actions it contains. The weights assigned to each action type reflect their relative importance in exploring new areas within the application.

Based on an analysis of the application's properties, behavior, and likelihood of exploring new areas, weight values were assigned to each action. The actions "click" and "long-click" were given a weight value of 3 as they are highly likely to reveal new areas of the application. Similarly, "text-entry" and "enter" have a high probability of uncovering new areas, resulting in an assigned weight value of 3 as well. The action "run-in-background" was also given a weight value of 3 as it has the potential to explore application features that may not be visible when the app is in the foreground.

Although the "check-uncheck" action has the potential to explore new areas, a weight value of 2 was assigned to it, considering the possibility that its probability of occurrence may be less than the actions mentioned above. As the "back" command returns the user to the previous screen, which can disclose undiscovered areas, a weight value of 2 was assigned

to it. The GUI actions "swipe-up", "swipe-down", "swipe-right", and "swipe-left" were assigned a lower weight value of 1. This decision was based on the understanding that these actions primarily manipulate the currently loaded screen and are not as likely to reveal new sections of the application. It's crucial to remember that the above weight distribution may not be ideal for every application. Depending on the specific requirements of the application, the weight values can be fine-tuned to get to obtain better results.

ECWCP techniques prioritize test cases by combining element coverage, test case cost, and the different importance of various action types within a test case in exploring new areas of the application under test. In the *ECWCP* approach, the total weight of a test case is calculated by adding the weights of the individual action types it contains. The weight assignments for individual actions are shown in Table 5.1. Additionally, the number of uncovered unique UI elements covered by the test case and their cost are computed. The four ECWCP strategies utilize the following equations to calculate prioritization scores:

$\boldsymbol{ECWC_TP_1}$ **:**

$$p\_score = \frac{weight * e\_count}{execution\_time} \tag{5.9}$$

$\boldsymbol{ECWC_LP_1}$ **:**

$$p\_score = \frac{weight * e\_count}{t\_length} \tag{5.10}$$

$\boldsymbol{ECWC_TP_2}$ **:**

$$p\_score = e\_count + \frac{weight}{execution\_time} \tag{5.11}$$

$\boldsymbol{ECWC_LP_2}$ **:**

$$p\_score = e\_count + \frac{weight}{t\_length} \tag{5.12}$$

5.1.4. Algorithm Overview

Algorithm 5 describes the pseudocode for the test prioritization algorithm. The algorithm takes the default order test suite T as input and produces the prioritized test suite T'. Initially, the algorithm sets *testcase_count* to total count of test cases in the test suite T and *selected_test_case* to 0 as the number of test cases chosen so far.

Algorithm 5: Prioritization Algorithm Pseudocode

```
   Input: Test suite T
   Output: Prioritized Test Suite T'
 1 testcase_count = total count of test cases in the test suite T
 2 selected_testcase_count = 0
 3 T' = []
 4 while selected_testcase_count ¡ testcase_count do
 5 |   current_max = -1
 6 |   current_best = Null
 7 |   for test case T_i in T do
 8 |   |   if T_i not in T' then
 9 |   |   |   compute element_count as the number of newly covered UI elements by
   |   |   |    T_i
10 |   |   |   compute weight of the test case T_i
11 |   |   |   compute tc_length as the length of T_i
12 |   |   |   get exec_time as the time required to execute T_i
13 |   |   |   compute prioritization score p_score for T_i
14 |   |   |   if p_score ¿ current_max then
15 |   |   |   |   current_best = T_i
16 |   |   |   |   current_max = p_score
17 |   |   |   end
18 |   |   |   else if pScore == current_max then
19 |   |   |   |   break tie randomly
20 |   |   |   end
21 |   |   end
22 |   end
23 |   add current_best to T'
24 |   remove current_best from T
25 |   Mark all unique UI elements covered by current_best as covered
26 |   selected_testcase_count += 1
27 end
28 return T'
```

It also initializes T' to an empty list. Then, a while loop is executed until all test cases in T are selected.

The algorithm iterates over all the test cases in test suite T using a for loop in each iteration of the while loop. The algorithm calculates the number of newly covered UI elements by test case T_i, the weight of T_i, the length of T_i, and the execution time required to run T_i in seconds for each test case T_i that has not been selected yet. Based on the specified prioritizing approach, these data are utilized to generate the prioritization score *p_score* for

T_i.

If the computed *p_score* is larger than the current maximum *current_max*, T_i is chosen as the current best test case, and its *p_score* becomes the new *current_max*. If the computed *p_score* equals the *current_max*, the algorithm randomly breaks the tie. After the for loop is completed, the algorithm adds the current best test case *current_best* to the prioritized test suite T', removes it from the original test suite T, and marks all unique UI elements covered by *current_best* as covered. It also increases the *selected_test_case* by one.

Once all of the test cases in T have been selected, the algorithm returns T' as output.

5.2. Empirical Analysis

This section outlines the experimental setup utilized to conduct empirical analysis by applying the proposed prioritization techniques on five subject applications to answer the following research questions:

5.2.1. Research Questions

RQ1. **Does the Element Coverage-based Prioritization (ECP) strategy achieve higher APSC and APBC scores compared to the default and random ordering?**

RQ2. **Does the Element Coverage and Cost-based Prioritization (ECCP) strategies achieve higher APSC and APBC scores compared to the default and random ordering?**

RQ3. **Does the Element Coverage and Weighted Cost-based Prioritization (ECWCP) strategies achieve higher APSC and APBC scores compared to the default and random ordering?**

RQ4. **Does prioritizing test cases based on their execution time compared to prioritizing test cases based on their length improve APSC and APBC scores?**

TABLE 5.3. Characteristics of subject applications

Application Name	# Lines Of Code	# Branches	# Methods	# Classes	#Activity
AnkiDroid	29063	11772	4091	500	21
Tricky Tripper	8244	2512	1766	290	16
The Kana Quiz	4453	2231	629	87	7
Tickmate	2654	770	395	60	10
SimpleReminder	1126	314	292	48	4

5.2.2. Subject Applications and Test Case Generation

Table 5.3 provides the characteristics of five subject applications: *AnkiDroid*, *Tricky Tripper*, *The Kana Quiz*, *Tickmate*, and *SimpleReminder*. These applications are open-source, written in Java, and downloaded from the F-droid [5] open-source application repository. *AnkiDroid* has the largest codebase among the tested Android applications, with 21 activities, 29063 lines of code (LOC), 11772 branches, 4091 methods, and 500 classes, indicating a large and more intricate program. *Tricky Tripper* also has a significant number of activities (16), lines of code (8244), branches (2512), methods (1766), and classes (290), but less than *AnkiDroid*. *The Kana Quiz* has fewer activities (7), lines of code (4453), branches (2231), methods (629), and classes (87) than both *AnkiDroid* and *Tricky Tripper*, indicating a simpler application. *Tickmate* and *SimpleReminder* have the lowest numbers of lines of code, branches, methods, and classes among the five applications, indicating that they are relatively simple applications. *Tickmate* has 10 activities, 2654 lines of code, 770 branches, 395 methods, and 60 classes, while *SimpleReminder* has 4 activities, 1126 lines of code, 314 branches, 292 methods, and 48 classes.

The reinforcement learning algorithm SARSA was used to create 10 test suites for each of the five Android applications. The algorithm was run for two hours to generate each test suite. For the *AnkiDroid* application, the test suites consist of 120 to 148 test cases, achieving a line coverage of 39.62% and a branch coverage of 25.16%. The number of activities covered ranges from 14 to 16. The test suites for *Tricky Tripper* include 106 to 125 test cases, with a line coverage of 36.61% and a branch coverage of 20.27%. The number of activities covered ranges from 8 to 10. *The Kana Quiz* application's test suites contain between 101 to 124 test cases, with a line coverage of 63.3% and a branch coverage of

46.71%. Only 6 activities are covered by the test suites. *Tickmate's* test suites consist of 94 to 116 test cases, achieving a line coverage of 78.02% and a branch coverage of 57.87%. The test suites cover 10 activities. Lastly, for the *SimpleReminder* application, the test suites contain between 106 to 125 test cases, with a line coverage of 67.18% and a branch coverage of 48.89%. The test suites cover 4 activities.

5.2.3. Experimental Setup and Evaluation Metric

The proposed strategies break the tie randomly when it encounters two (or more) test cases with the same prioritization score. To mitigate the variance due to random tie-breaking, each strategy was applied 10 times on every test suite of all the subject applications. The performance of the proposed strategies is compared with default order and random order in terms of two metrics:: APSC (Average Percentage of Statement Coverage) and APBC (Average Percentage of Branch Coverage). Inspired by APFD (Average Percentage of Faults Detected), APSC and APBC metrics are introduced by Li et al. [74] to measure the coverage rate of a test suite. For a test suite containing n test cases, APSC and APBC scores are calculated by the following equations:

$$APSC = 1 - \frac{TS_1 + TS_2 + ... + TS_m}{nm} + \frac{1}{2n}$$

$$APBC = 1 - \frac{TB_1 + TB_2 + ... + TB_m}{nm} + \frac{1}{2n}$$

where, m represents the total number of statements or branches covered by the test suite, TS_i, and TB_i refer to the initial test case that encounters the statement or branch i.

A Python script is used to calculate APSC and APBC scores from the coverage report for every prioritized test suite of an application for a strategy and report the average scores.

5.3. Results and Discussion

Table 5.4 shows the APSC and table 5.5 shows the APBC scores achieved by the proposed strategies as well as the default and random ordering. All of the proposed techniques exhibited superior performance compared to both the default and randomly ordered

strategies, resulting in better APSC and APBC scores. The significance of the improvement in APSC and APBC scores attained by the proposed strategies, in comparison to default and random order, was examined through a statistical analysis employing the t-test. The outcomes of the t-test indicate a remarkably significant difference ($p < 0.001$) in APSC and APBC scores across all the proposed techniques, when compared to both the default and randomly ordered test suites.

TABLE 5.4. APSC scores by applications

Apps	Default	Random	ECP	ECC_TP_1	ECC_LP_1	ECC_TP_2	ECC_LP_2	$ECWC_TP_1$	$ECWC_LP_1$	$ECWC_TP_2$	$ECWC_LP_2$
AnkiDroid	92.78	94.11	98.44	96.86	96.23	98.29	98.31	98.50	98.48	**98.55**	98.54
Tricky Tripper	93.03	94.59	99.34	98.98	98.65	99.33	99.32	99.39	99.39	**99.44**	99.44
The Kana Quiz	94.83	94.78	97.90	96.30	95.65	97.62	97.61	**98.11**	98.06	98.02	97.95
Tickmate	88.86	90.01	97.67	95.99	94.96	97.43	97.44	97.63	97.61	**97.71**	97.66
SimpleReminder	92.92	93.62	97.61	96.63	95.73	97.45	97.46	**97.74**	97.59	97.66	97.54

TABLE 5.5. APBC scores by applications

Apps	Default	Random	ECP	ECC_TP_1	ECC_LP_1	ECC_TP_2	ECC_LP_2	$ECWC_TP_1$	$ECWC_LP_1$	$ECWC_TP_2$	$ECWC_LP_2$
AnkiDroid	94.91	95.87	98.96	97.82	97.34	98.83	98.84	98.99	98.98	**99.05**	99.03
Tricky Tripper	95.53	96.58	99.69	99.40	99.17	99.66	99.65	99.71	99.72	**99.77**	99.76
The Kana Quiz	94.76	95.10	98.44	96.99	96.40	98.28	98.25	**98.61**	98.58	98.60	98.52
Tickmate	90.19	90.82	97.59	95.92	95.05	97.07	97.08	97.65	97.54	**97.75**	97.61
SimpleReminder	92.09	92.80	97.53	96.03	95.05	97.20	97.20	**97.69**	97.49	97.67	97.47

RQ1: Does the Element Coverage-based Prioritization (ECP) strategy achieve higher APSC and APBC scores compared to the default and random ordering?

The **Element Coverage-based Prioritization (*ECP*)** technique outperformed the default and random ordering methods, consistently earning considerably better APSC and APBC scores for all applications investigated in this study. Table 5.6 presents the improvement in APSC and APBC scores resulting from the ECP approach as compared to default and random methods.

According to the results, the *ECP* strategy improved the APSC and APBC scores of the *AnkiDroid* application by 5.66% and 4.05%, respectively, compared with the default ordering method. Additionally, the ECP strategy yielded 4.33% higher APSC and 3.09% higher APBC scores than the random ordering strategy for *AnkiDroid*.

In comparison to the default ordering, the *ECP* technique resulted in a considerable boost in *Tricky Tripper*'s APSC and APBC scores of 6.31% and 4.16%, respectively. The

ECP strategy improved *Tricky Tripper*'s APSC and APBC by 4.75% and 3.11%, respectively, as compared to random ordering.

TABLE 5.6. APSC and APBC improvement over default and random by ECP

	APSC		**APBC**	
Apps	ΔDefault	ΔRandom	ΔDefault	ΔRandom
AnkiDroid	5.66	4.33	4.05	3.09
Tricky Tripper	6.31	4.75	4.16	3.11
The Kana Quiz	3.07	3.12	3.68	3.34
Tickmate	8.81	7.66	7.40	6.77
SimpleReminder	4.69	3.99	5.44	4.73

When compared to *The Kana Quiz* application's default ordering method, the *ECP* technique increased the APSC and APBC scores by 3.07% and 3.68%, respectively. For the same application, the APSC and APBC are improved by 3.12% and 3.34%, respectively, as compared to the random ordering technique.

In the instance of *Tickmate*, the *ECP* technique increased APSC and APBC scores by 8.81% and 7.40%, respectively, over the default and by 7.66% and 6.77%, respectively, over random ordering.

With APSC and APBC scores improving by 4.69% and 5.44%, respectively, the *ECP* method surpassed the default ordering for *SimpleReminder*. It also outperformed random ordering by 3.99% and 4.73%, respectively.

RQ2: Does the Element Coverage and Cost-based Prioritization (ECCP) strategy achieve higher APSC and APBC scores compared to the default and random ordering?

The **Element Coverage and Cost-based Prioritization (*ECCP*)** strategies prioritize critical test cases early by combining coverage of unique UI elements with the cost of executing test cases to optimize resource usage and time. Tables 5.7 and 5.8 demonstrate the APSC and APBC improvement for five applications by the proposed *ECCP* approaches, namely ECC_TP_1, ECC_LP_1, ECC_TP_2 and ECC_LP_2.

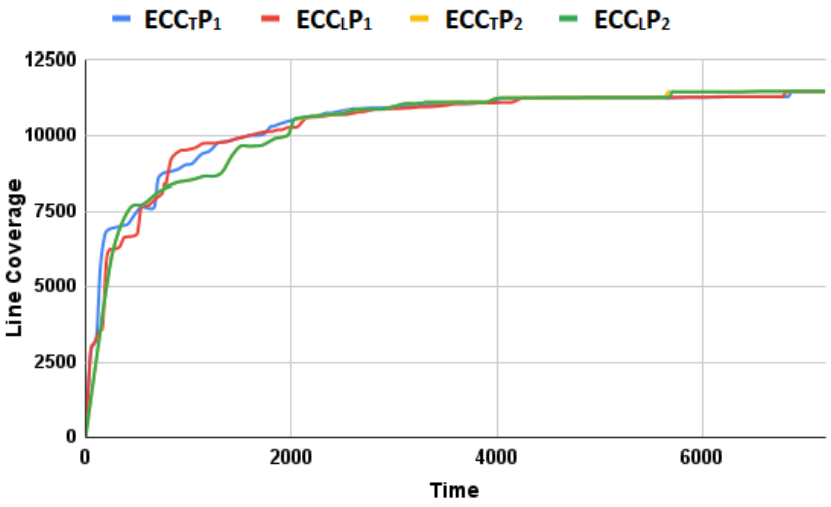
TABLE 5.7. APSC improvement over default and random by ECCP strategies

	ECC_TP_1		ECC_LP_1		ECC_TP_2		ECC_LP_2	
Apps	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom
AnkiDroid	4.08	2.75	3.45	2.12	5.51	4.18	5.53	4.20
Tricky Tripper	5.95	4.39	5.62	4.06	6.30	4.74	6.29	4.73
The Kana Quiz	1.47	1.52	0.82	0.87	2.79	2.84	2.78	2.83
Tickmate	7.13	5.98	6.10	4.95	8.57	7.42	8.58	7.43
SimpleReminder	3.71	3.01	2.81	2.11	4.53	3.83	4.54	3.84

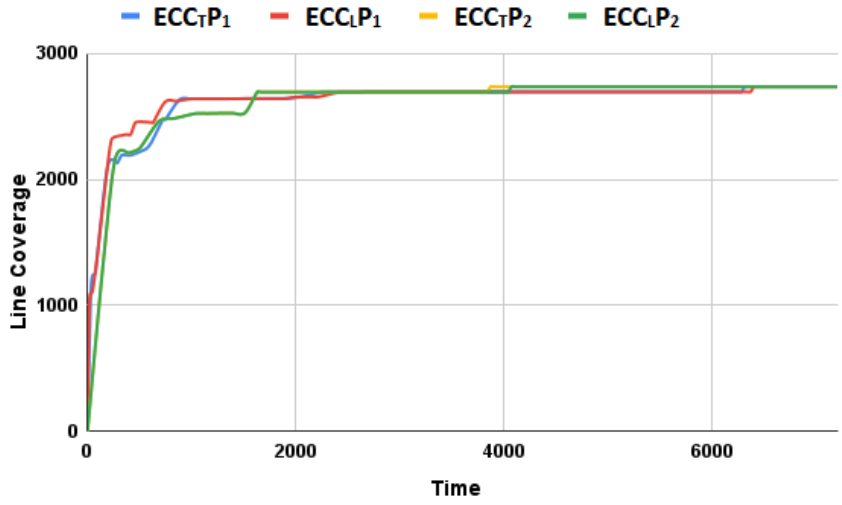
TABLE 5.8. APBC improvement over default and random by ECCP strategies

	ECC_TP_1		ECC_LP_1		ECC_TP_2		ECC_LP_2	
Apps	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom
AnkiDroid	2.91	1.95	2.43	1.47	3.92	2.96	3.93	2.97
Tricky Tripper	3.87	2.82	3.64	2.59	4.13	3.08	4.12	3.07
The Kana Quiz	2.23	1.89	1.64	1.30	3.52	3.18	3.49	3.15
Tickmate	5.73	5.10	4.86	4.23	6.88	6.25	6.89	6.26
SimpleReminder	3.94	3.23	2.96	2.25	5.11	4.40	5.11	4.40

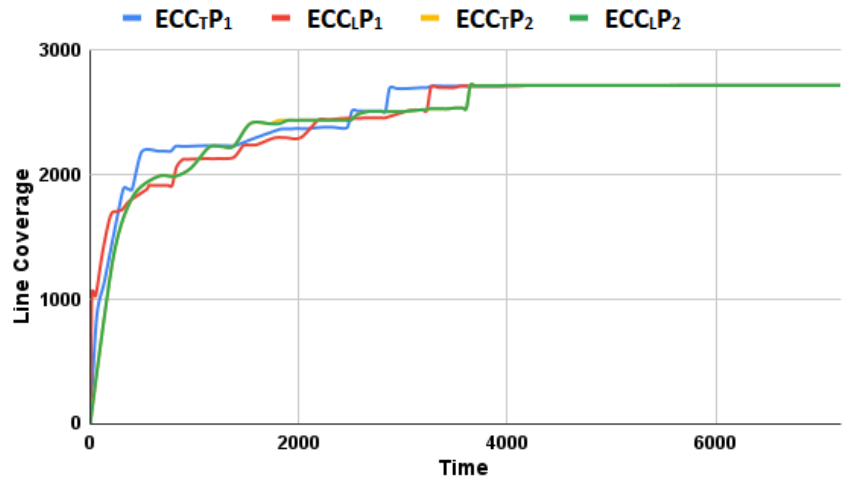
The results indicate that the enhancement in APSC for ECC_TP_1 displays a range of 1.47% to 7.13% as opposed to the default order, and a range of 1.52% to 5.98% when compared with random ordering. The ECC_TP_1 technique results in varied enhancements to APBC in comparison to the default order, ranging from 2.23% to 5.73%. Additionally, when compared to the random ordering prioritization, the range of improvement is between 1.89% to 5.10%. ECC_LP_1 improved the APSC by 0.82% to 6.10% compared to the default ordering and by 0.87% to 4.95% compared to random ordering. In contrast to the default order, the corresponding improvement in APBC ranged from 1.64% to 4.86%. Likewise, the resulting APBC improvement compared to random ordering was observed to range between 1.30% and 4.23%. The results also illustrate that the improvement in APSC for ECC_TP_2 showed a range of values, ranging from 2.79% to 8.57% compared to the default ordering, and from 2.84% to 7.42% when compared to random ordering. Regarding the default ordering, this strategy improves APBC by a range between 3.52% and 6.88%, and compared to random ordering, it ranges between 2.96% and 6.25%. ECC_LP_2 strategy yielded APSC improvements ranging from 2.78% to 8.58% relative to the default ordering, and from 2.83% to 7.43% relative to random ordering. Compared to the default and random ordering, ECC_LP_2 exhibited variations in the ranges of 3.49% to 6.89% and 2.97% to 6.26%, respectively, in terms of the reported increase in APBC.



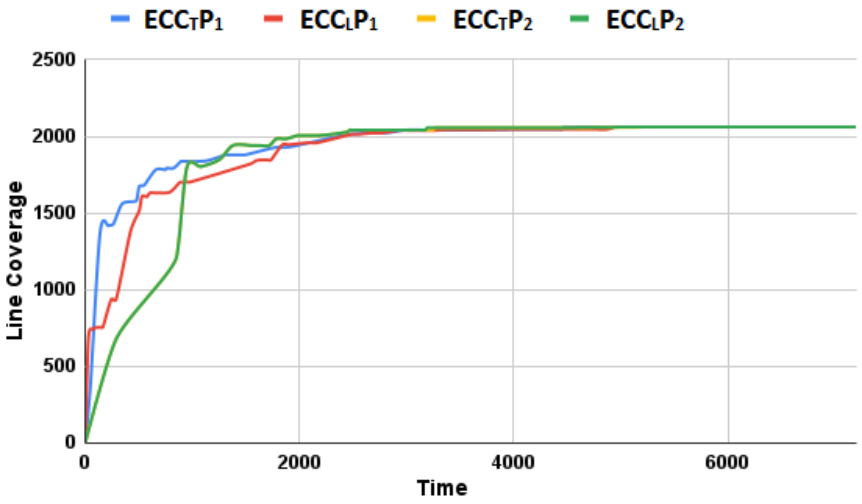
(A) AnkiDroid: Coverage Progress Over Time



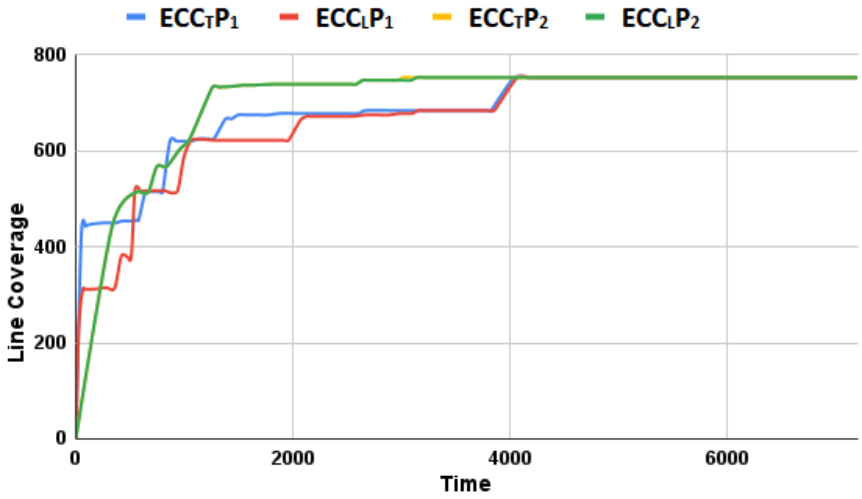
(B) Tricky Tripper: Coverage Progress Over Time



(C) The Kana Quiz: Coverage Progress Over Time



(D) Tickmate: Coverage Progress Over Time



(E) SimpleReminder: Coverage Progress Over Time

FIGURE 5.1. ECCP strategies Coverage Progress Over Time

The line coverage progress over time by different ECCP strategies for all the subject applications is plotted in Figure 5.1 to investigate if there are any significant differences. The X-axis represents time in seconds, and the Y-axis represents the number of lines covered. Either ECC_TP_1 or ECC_LP_1 exhibits an initial surge, yet their advancement lacks consistency. Conversely, the progress for all applications by ECC_TP_2 and ECC_LP_2 exhibit a close resemblance. Despite a relatively modest pace of advancement initially compared to $ECCP_1$ strategies, $ECCP_2$ strategies tend to attain maximal coverage prior to $ECCP_1$ strategies in

the majority of applications.

In general, the performance of $ECCP$ strategies was better than default and random ordering, but it was not as strong as the other prioritization strategies investigated in this study. In terms of APSC and APBC scores, both ECC_TP_2 and ECC_LP_2 displayed superior progress when compared to ECC_TP_1 and ECC_LP_1, and were able to attain nearly similar results as ECP. These results are not surprising because ECC_TP_1 and ECC_LP_1 strategies prioritize the test cases based on the coverage per unit cost. This way a test case will get a low prioritization score if the cost(test case length or execution time) is high even when it covers a high number of UI elements. On the other hand, ECC_TP_2 and ECC_LP_2 strategies give more importance to the element coverage and add a penalty based on the test case cost. None of these strategies consider the test case complexity and therefore lower APSC and APBC scores are expected.

RQ3: Does the Element Coverage and Weighted Cost-based Prioritization (ECWCP) strategy achieve higher APSC and APBC scores compared to the default and random ordering?

The study investigated four distinct **Element Coverage and Weighted Cost-based Prioritization (*ECWCP*)** strategies: $ECWC_TP_1$, $ECWC_LP_1$, $ECWC_TP_2$, and $ECWC_LP_2$. In addition to the element coverage and the cost, these strategies consider the importance of action types to explore new areas of the application. The employment of the aforementioned strategies has led to a noteworthy increase in both APSC and APBC scores across all the subject applications. The APSC improvements by $ECWCP$ strategies are shown in Table 5.9 and the corresponding APBC improvements by the $ECWCP$ strategies are shown in Table 5.10.

The $ECWC_TP_1$ method consistently outperformed default order, random order, ECP, and all the $ECCP$ strategies for all the subject applications. For *The Kana Quiz* and *SimpleReminder*, this strategy outperformed other $ECWCP$ strategies too, and achieved the highest APSC and APBC scores. The APSC improvement for this strategy ranges from 3.28% to 8.77% over the default, and from 3.33% to 7.62% over random ordering.

TABLE 5.9. APSC improvement over default and random by ECWCP strategies

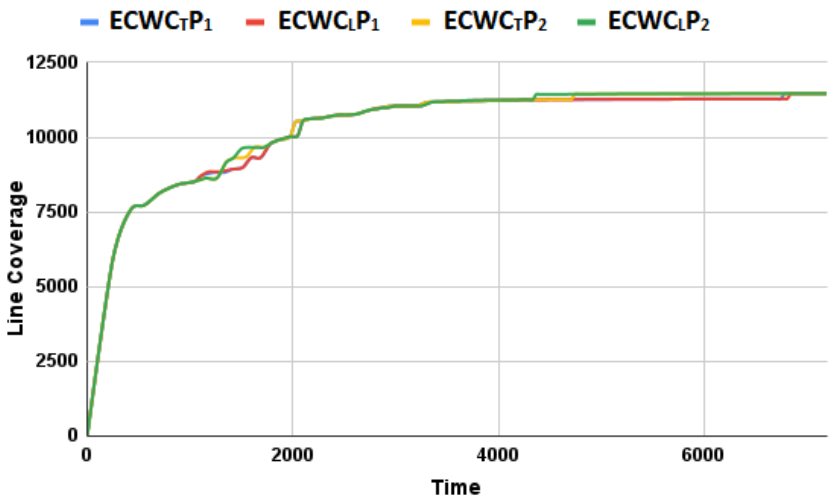
	$ECWC_TP_1$		$ECWC_LP_1$		$ECWC_TP_2$		$ECWC_LP_2$	
Apps	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom
AnkiDroid	5.72	4.39	5.70	4.37	5.77	4.44	5.76	4.43
Tricky Tripper	6.36	4.80	6.36	4.80	6.41	4.85	6.41	4.85
The Kana Quiz	3.28	3.33	3.23	3.28	3.19	3.24	3.12	3.17
Tickmate	8.77	7.62	8.75	7.60	8.85	7.70	8.80	7.65
SimpleReminder	4.82	4.12	4.67	3.97	4.74	4.04	4.62	3.92

TABLE 5.10. APBC improvement over default and random by ECWCP strategies

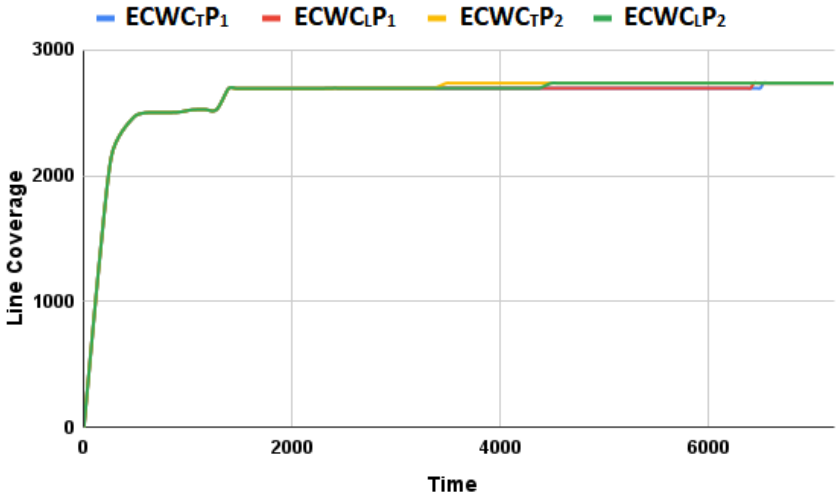
	$ECWC_TP_1$		$ECWC_LP_1$		$ECWC_TP_2$		$ECWC_LP_2$	
Apps	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom	ΔDefault	ΔRandom
AnkiDroid	4.08	3.12	4.07	3.11	4.14	3.18	4.12	3.16
Tricky Tripper	4.18	3.13	4.19	3.14	4.24	3.19	4.23	3.18
The Kana Quiz	3.85	3.51	3.82	3.48	3.84	3.50	3.76	3.42
Tickmate	7.46	6.83	7.35	6.72	7.56	6.93	7.42	6.79
SimpleReminder	5.60	4.89	5.40	4.69	5.58	4.87	5.38	4.67

As for APBC, the improvement varies from 3.85% to 7.46% over default, and from 3.12% to 6.83% when compared to random ordering. The $ECWC_LP_1$ technique is another strategy that consistently outperformed default order, random order, ECP, and all the $ECCP$ strategies. Among the $ECWCP$ strategies, it achieved the lowest or the second lowest scores. The APSC gain for $ECWC_LP_1$ ranges from 3.23% to 8.75% in contrast to default, and from 3.28% to 7.60% compared to random ordering. The resulting APBC improvement ranged from 3.82% to 7.35% over default ordering, and from 3.11% to 6.72% over random ordering.

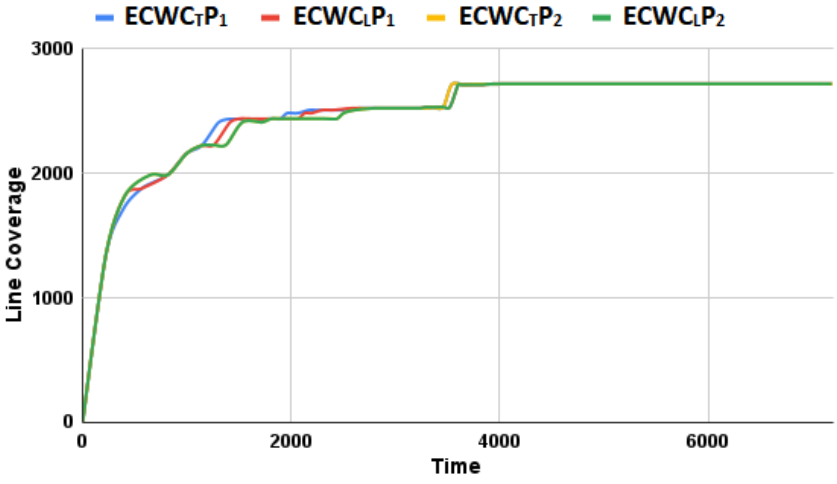
$ECWC_TP_2$ strategy outperformed all other proposed strategies and achieved the highest APSC and APBC scores for three applications: *AnkiDroid*, *Tricky Tripper* and *Tickmate*. Moreover, it achieved the second-highest APSC and APBC scores for *SimpleReminder*, and the second-highest APBC but third-highest APSC score for *The Kana Quiz*. When compared to the default order, it amplified APSC by 3.19% to 8.85%, while the improvement is 3.24% to 7.70% over random ordering. Similarly, APBC improvement ranges from 3.84% to 7.56% over default order, and 3.18% to 6.93% over random ordering. $ECWC_LP_2$ is found to be an effective prioritization strategy with consistently achieving second or third-highest APSC and APBC scores compared to the other strategies.



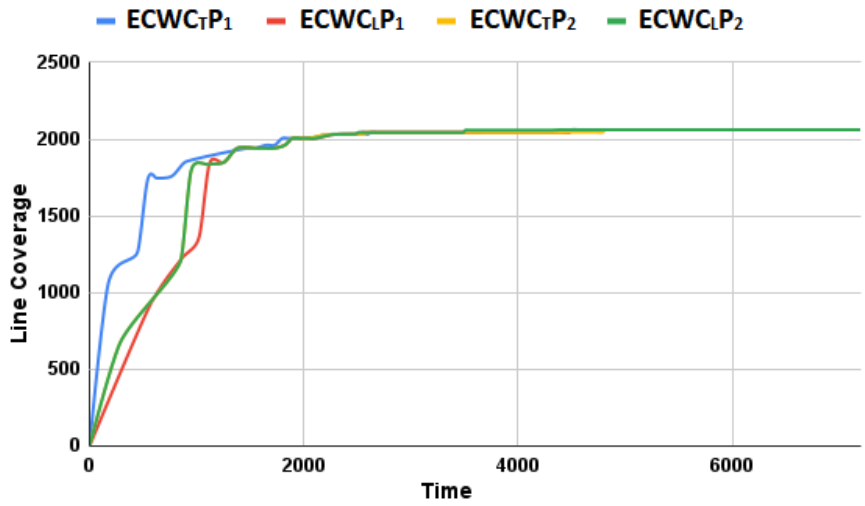
(A) AnkiDroid: Coverage Progress Over Time



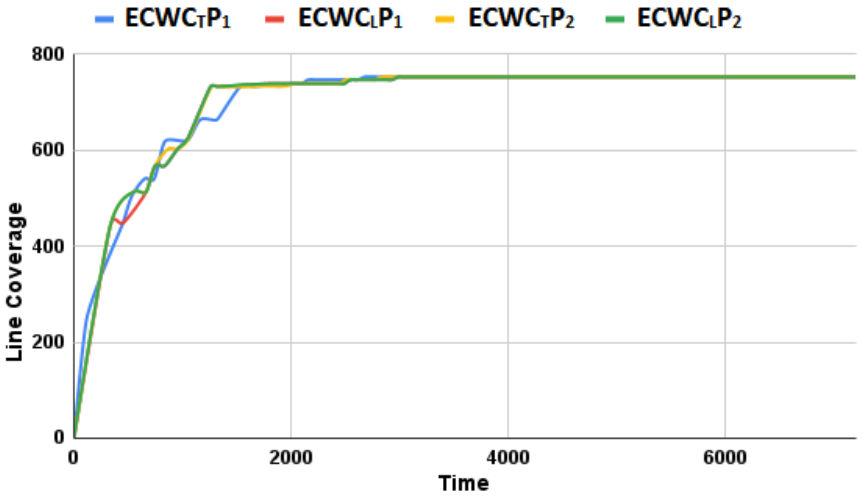
(B) Tricky Tripper: Coverage Progress Over Time



(C) The Kana Quiz: Coverage Progress Over Time



(D) Tickmate: Coverage Progress Over Time



(E) SimpleReminder: Coverage Progress Over Time

FIGURE 5.2. ECWCP strategies Coverage Progress Over Time

The APSC improvement was between 3.12% to 8.80% over default ordering and 3.17% to 7.65% over random ordering. The improvement in APBC ranged from 3.76% to 7.42% over default ordering, and from 3.16% to 6.79% over random ordering.

Figure 5.2 illustrates the line coverage progress over time by *ECWCP* strategies for all the subject applications. The X-axis represents time in seconds, and the Y-axis represents the number of lines covered. For *AnkiDroid*, *Tricky Tripper*, and *The Kana Quiz* applications, the progress is identical at the beginning for all the *ECWCP* strategies. Some

variations are observed in the later stages, but they are not very significant. $ECWCP_1$ exhibits an early jump in line coverage for *Tickmate*, but after some time, the progress looks almost the same for all the strategies. In the case of *SimpleReminder*, there are some variations early on, but the progress becomes the same for all the strategies in the later stages. Overall, it seems that all the $ECWCP$ strategies achieve the highest coverage at a similar time, and the lines overlap each other.

The findings of the study indicate that the weighting mechanism used in the Element Coverage and Weighted Cost-based Prioritization($ECWCP$) techniques can significantly increase the value of code coverage measures such as APSC and APBC. The weights assigned to various action types help to differentiate between the complexity and significance of different actions. For example, clicks and long-clicks may be more important than swipes, and the weighting mechanism enables the prioritization technique to consider these differences. If two test cases have different action sets but have the same length or take the same time to execute, the test case with more important actions, such as clicks and long-clicks, will receive higher priority. This way, it is possible to discover the complex test cases and critical paths that improve the APSC and APBC scores.

RQ4: Does prioritizing test cases based on their execution time compared to prioritizing test cases based on their length improve APSC and APBC scores?

The study attempted to investigate whether the choice of test case length versus execution time for computing the prioritizing score has any effect on the proposed strategies. The performance of ECC_TP_1 was analyzed compared to ECC_LP_1, ECC_TP_2 was analyzed compared to ECC_LP_2, $ECWC_TP_1$ was analyzed compared to $ECWC_LP_1$, and $ECWC_TP_2$ was analyzed compared to $ECWC_LP_2$. In most instances, subtle variations exist without marked significance, except for ECC_TP_1 as shown in table 5.11 and 5.12.

TABLE 5.11. APSC: Time vs Length

Apps	ECC_TP_1 - ECC_LP_1	ECC_TP_2 - ECC_LP_2	$ECWC_TP_1$ - $ECWC_LP_1$	$ECWC_TP_2$ - $ECWC_LP_2$
AnkiDroid	0.63	-0.02	0.02	0.01
Tricky Tripper	0.33	0.01	0	0
The Kana Quiz	0.65	0.01	0.05	0.07
Tickmate	1.03	-0.01	0.02	0.05
SimpleReminder	0.90	-0.01	0.15	0.12

TABLE 5.12. APBC: Time vs Length

Apps	ECC_TP_1 - ECC_LP_1	ECC_TP_2 - ECC_LP_2	$ECWC_TP_1$ - $ECWC_LP_1$	$ECWC_TP_2$ - $ECWC_LP_2$
AnkiDroid	0.48	-0.01	0.01	0.02
Tricky Tripper	0.23	0.01	-0.01	0.01
The Kana Quiz	0.59	0.03	0.03	0.08
Tickmate	0.87	-0.01	0.11	0.14
SimpleReminder	0.98	0	0.20	0.20

ECC_TP_1 achieved better APSC and APBC scores than ECC_LP_1 with APSC differences ranging from 0.33% to 1.03% and APBC differences ranging from 0.23% to 0.98%. In the case of ECC_TP_2 vs ECC_LP_2, the difference ranges from -0.01% to 0.02% for APSC and -0.1% to 0.3% for APBC. Similarly, $ECWC_TP_1$ versus $ECWC_LP_1$ did not show a significant advantage of using one over another with APSC differences ranging from 0 to 0.15% and APBC differences ranging from -0.01% to 0.20%. The APSC and APBC differences for $ECWC_TP_2$ and $ECWC_LP_2$ respectively range from 0 to 0.12% and 0.01% to 0.20%.

5.4. Threats to Validity

This study has demonstrated the potential of prioritization strategies based on element coverage, test case cost, and test case weight. However, there are several threats to consider as this work may not generalize to all applications and test suites.

Internal Validity: This work compares the rate of code coverage for the techniques using APSC and APBC. The results may be affected by the test case generation method and the quality of the test cases. To mitigate this threat, 10 suites of different lengths were used for each application by employing random tie-breaking. Also, each strategy was run 10 times on each test suite to calculate the average score for each strategy for each application.

External Validity: The results of this study may not be applicable to other platforms and languages as the examination was limited to the Android domain, specifically focusing on unique Android UI elements and action types. Additionally, the empirical analysis was performed on only five Android applications, which may not represent the entire Android population. To mitigate this potential limitation, five Android applications of different purposes were selected, each with varying numbers of lines of code, branches, methods, classes, and activities.

Construct Validity: The cost of test cases may vary by project, team, and other factors. For instance, there may be costs associated with cloud services, servers, employees participating in the testing process to evaluate results, and more. For this work, test case length and execution time were used as a surrogate for cost without considering other possible costs, given the fully automated process, including code coverage calculations. The weight calculations used for different action types may change for other applications but are sensible in this study as the weight of actions typically correlated with the amount of code access, i.e., selecting a checkbox calls less code than clicking a *submit* button in most of the applications used in this study. This threat was minimized by carefully understanding the applications under test and assigning weights respectively. Other applications may have different characteristics where different weight values may be tailored respectively.

Conclusion Validity: The performance of the proposed strategies was evaluated by comparing APSC and APBC scores with the test suite's original order and randomly ordered prioritization. This choice was made to assess the effectiveness of the proposed techniques in terms of enhancing the code coverage rate. However, other work could examine different metrics such as fault-finding effectiveness.

CHAPTER 6

CONCLUSIONS

6.1. Summary and Conclusions

Android applications are event-driven systems that often have a large event space. The exponential number of event combinations poses challenges to testing budgets. This study presents a novel approach of test case generation using reinforcement learning algorithms to generate test cases with high code coverage. Additionally, it optimizes the effectiveness of the test suites with test suite prioritization techniques for Android automated GUI (Graphical User Interface) tests generated by reinforcement learning algorithms. The fundamental goal of this hybrid approach is to improve testing efficiency by not only automating test case generation but also by utilizing test case prioritization strategies to increase the rate of code coverage during test execution. Reinforcement learning-based test generation algorithm uses trial-and-error interactions to optimize event selection and systematically explore an AUT's GUI to generate test cases. The study adopted two popular reinforcement learning algorithms Q-learning and SARSA, to systematically explore Android application GUI space and thereby generate test cases in the form of a sequence of events.

Empirical evaluation shows that for the same set of test generation parameters (i.e. generation time, the delay between events, home button probability, etc.), the Q-learning-based technique achieves 10.30% higher block coverage on average than random test generation, with a range of 3.31% to 18.83% improvement across eight subject applications.

The performance of SARSA is evaluated compared to the popular test generation tool Monkey for seven native Android applications. SARSA outperformed Monkey for all the subject applications in line, branch, method, and class coverage for the same set of test parameters. Even though the SARSA-based technique proposed in this study does not support system events as Monkey does, SARSA achieved 9.87% to 24.79% better line coverage, 6.9% to 20.09% better branch coverage, 7.88% to 28.48% better method coverage, and 3.74% to 35.02% better class coverage than Monkey. SARSA consistently achieves higher

minimum, median, and maximum coverage than Monkey for all runs across all the subject applications. Close inspection of the coverage progress over time shows that SARSA achieves higher code coverage than Monkey and does this at a faster rate of coverage.

Test case prioritization aims to enhance testing efficiency through productive reordering. Although SARSA-generated test cases demonstrated a notable level of code coverage, this study recognizes the potential for enhancing the efficacy of the test suite and devised prioritization techniques for the purpose of improving code coverage rates. This dissertation work introduced four prioritization strategies, namely Pairwise Event Interaction Coverage, Pair-Activity Coverage (PA), Pair-State Coverage (PS), and Pair-State-Activity Coverage (PSA) prioritization, by integrating pairwise interaction, application state coverage, and activity coverage. The pairwise approach assigns priority to test cases on the basis of the number of pair interactions present within each of them. The PA strategy effectively integrates unique activity coverage with pairwise interaction coverage. In contrast, the PS approach encompasses pairwise interaction coverage alongside unique state coverage. Additionally, the PSA strategy encompasses the combination of all three aforementioned factors.

The results of an empirical analysis show that each prioritization algorithms significantly outperform default and random orderings for the test suites, with activity coverage-based prioritization PA and PSA outperforming pairwise and PS. PA achieved the highest improvement for *Trickytripper*, *The Kana Quiz*, and *Tickmate* with an average increase of 6.24%, 3.24%, and 8.14% in APSC and 4.14%, 3.94% and 7.32% in APBC over default ordering and 4.68%, 3.30% and 7.00% in APSC and 3.09%, 3.60% and 6.69% in APBC over random ordering. PSA achieved the best improvement for *Ankidroid* and *SimpleReminder* with an average increase of 5.54% and 5.17% in APSC and 4.01% and 6.26% in APBC over default ordering, and 4.21% and 4.47% in APSC and 3.05% and 5.55% in APBC over random ordering.

Graphical User Interface (GUI) of Android applications provides functionalities to the end users. The superior performance achieved by PA and PSA strategies suggests a correlation between GUI element coverage and code coverage. Test case cost is a crucial

factor that has been used in several other prioritization techniques found in the literature. This study introduces a novel concept of "test case weight" and developed three new sets of prioritization strategies by combining "test case weight" with GUI element coverage and test case cost. These strategies are called Element Coverage-based Prioritization (ECP), Element Coverage and Cost-based Prioritization ($ECCP$), and Element Coverage and Weighted Cost-based Prioritization ($ECWCP$). Test case weight measures the complexity of a test case based on the importance of different action types. The hypothesis behind test case weight is that not all actions in a test case are equally important; some actions are more valuable for exploring new areas of the application. Test case cost was estimated by considering two factors: test case length and execution time.

An empirical study applies the above strategies to five Android applications to assess their effectiveness by comparing APSC (Average Percentage of Statement Coverage) and APBC (Average Percentage of Branch Coverage) rates of the prioritized test suites to the original order of the test suite and random order prioritization. The ECP strategy prioritizes test cases based solely on the coverage of UI elements, with no consideration for other factors. This means that the test case with the highest count of unique UI elements is assigned the highest priority. The ECP strategy outperforms both default and random order prioritization. It achieves 3.07% - 8.81% better APSC than the default order, and 3.12% - 7.66% better APSC score than the random order. Furthermore, ECP improves APBC by 3.68% - 7.40% compared to default, and 3.09% to 6.77% compared to random ordering.

The $ECCP$ strategy takes into account the cost of a test case, which is determined by its length and execution time, along with element coverage. However, it does not incorporate the complexity of the test case. This work examined four $ECCP$ strategies, namely ECC_TP_1, ECC_LP_1, ECC_TP_2, and ECC_LP_2 based on the way of calculating prioritization scores. These strategies perform better than the default and random order but worse than the ECP and $ECWCP$ strategies. Overall, the $ECCP$ strategies achieve 0.82% - 8.58% better APSC and 1.64% - 6.89% better APBC than the default order. They also achieve 0.87% - 7.43%

better APSC and 1.30% - 6.26% better APBC than random ordering.

The $ECWCP$ strategies assign weight values to different actions in a test case based on their importance for exploring new areas. $ECWCP$ strategies determine the total weight of a test case by summing the weights of the individual actions within the test case. The four $ECWCP$ strategies include: ($ECWC_TP_1$, $ECWC_LP_1$, $ECWC_TP_2$, and $ECWC_LP_2$) combine test case weight with element coverage and cost to identify a crucial path in maximizing coverage rate. These strategies achieve superior APSC and APBC scores compared not only to the default and random but also to other strategies (ECP and $ECCP$) proposed in this study. The APSC improvement by $ECWCP$ strategies compared to the default order ranges from 3.12% to 8.85%, and from 3.17% to 7.70% compared to the random order. Similarly, the APBC improvement by $ECWCP$ strategies ranges from 3.76% - 7.56% and 3.11% to 6.93% compared to the default and random order, respectively.

6.2. Implications

The results of this study have significant implications for the testing of Android applications. The proposed hybrid approach of test case generation and prioritization addresses the challenges posed by the exponential event combinations, aiming to enhance testing efficiency and improve software quality. Results indicate that the application of reinforcement learning algorithms, specifically Q-learning and SARSA, during the test case generation proved to be highly effective. The generated test cases effectively provided thorough testing coverage by covering a variety of GUI interactions and scenarios. The application of test case prioritization techniques allowed for the reordering of test cases, further improving efficiency by focusing on the most impactful test scenarios.

Overall, the findings suggest that the hybrid approach of reinforcement learning-based test case generation and prioritization may significantly enhance the efficiency and effectiveness of testing Android applications. By systematically exploring the GUI space, optimizing event selection, and prioritizing test cases, the proposed techniques contribute to improving software reliability and quality despite the challenges posed by complex event-driven systems.

CHAPTER 7

FUTURE WORK

This dissertation presents a hybrid methodology to automatically test Android applications. The proposed approaches utilize reinforcement learning to automate test case generation with the goal to obtain high code coverage. Furthermore, this work employs test case prioritization strategies based on multiple factors to prioritize reinforcement learning generated test cases and enhance code coverage rate. The results of the empirical investigations guide future work in the following areas:

7.1. Q-learning and SARSA with Varied Hyperparameter Settings:

The present study implemented Q-Learning and SARSA algorithms for Android GUI test generation and evaluated their performance by comparing them with random test generation. Selecting parameters for machine learning algorithms is always challenging. The choice of the learning rate, discount factor, event selection heuristics, and initial Q-value may affect the test generation process and show different results. The Q-learning-based test generation in this study adopts a greedy event selection policy, and SARSA utilizes ϵ-greedy policy for event selection with an ϵ value of 0.3 in an effort to balance exploration vs. exploitation. The greedy method always selects the event with the highest Q-value. As a result, exploration can get stuck in local optima. ϵ-greedy method occasionally selects random events with the ϵ value probability. Commonly used epsilon values found in the literature are 0.1 [68], 0.2 [44], 0.5 [71]. An improved version of the ϵ-greedy method called decayed-ϵ-greedy utilized [75] [14] [69] in many existing studies. This approach applies a high epsilon value initially to encourage early exploration and decay at each iteration.

A learning rate of 1 is set in the Q-learning and SARSA implementations to maximize the learning process. The learning rate α regulates how the Q-value will be updated to find the optimal policy. 0.001 [14], 0.02 [30], 0.5 [75] [126] and 1 [82] [35] are some of the common learning rate values used in the existing literature. A small learning rate gives more importance to the old value and updates the Q-Value in small steps. A high learning rate

gives more importance to new information and updates the Q-values in larger steps. The high learning rate utilized in the algorithms presented in this dissertation may not converge to the optimal policy. The decayed learning rate used in some research [65] can stabilize the output converges to the optimal policy.

The discount factor controls the extent to which expected future rewards affect the choice of an event in the current GUI state. A high discount factor encourages the selection of events that potentially lead to high rewards in future states. A low discount factor prioritizes immediate rewards over long-term rewards. Usually, the discount factor is set to as close to 1 as possible to ensure the future reward is not neglected. The most common discount factor value used in the literature is 0.9 [82] [69] [71] [126]. This study used a variable discount factor calculated by an exponential decay function based on an intuition that the agent should look further ahead (i.e., use a high discount value) when it encounters states with a small number of events. Future work will explore Q-learning and SARSA for Android test generation with different event selection heuristics, reward functions, and varied hyperparameter settings.

7.2. Test Generation with Other Reinforcement Learning Algorithms

This dissertation employs reinforcement learning algorithms, namely Q-learning and SARSA, to systematically explore the graphical user interface (GUI) of Android applications with the aim of automatically generating test cases. There are alternative reinforcement learning algorithms that may serve as viable options for generating Android test cases through the exploration of an application's GUI. Two potential algorithms future studies can investigate for Android GUI test generation:

Monte Carlo Tree Search (MCTS): MCTS is considered one of the best algorithms for exploration. It is particularly popular in domains with large state space such as board games and strategic planning. MCTS combines tree search with reinforcement learning to efficiently explore the space of possible actions and make informed decisions. The fundamental concept underlying the Monte Carlo Tree Search (MCTS) algorithm is to progressively construct a search tree through iterative simulation and evaluation of potential actions. The structure of

the tree is composed of discrete elements known as nodes, which signify the different states present within the search space. The nodes are connected by edges that denote the actions or pathways linking them together. The algorithm undergoes a series of four key steps during each iteration:

(1) **Selection:** The algorithm initiates at the root node and proceeds through the tree by adhering to a selection policy that typically balances exploration and exploitation.

(2) **Expansion:** When the algorithm encounters a node with unexplored actions, it expands the tree by adding child nodes that correspond to the available actions within that node.

(3) **Simulation:** The algorithm conducts a Monte Carlo simulation from the newly added node, by selecting actions until a terminal state or a predefined depth is reached. The simulation is conducted in accordance with a predetermined or random policy.

(4) **Backpropagation:** The simulation results are back propagated upward along the tree structure, thereby revising the statistical attributes assigned to all nodes that were traversed. This data guides the selection policy in subsequent iterations.

By iteratively applying these steps over a significant number of cycles, Monte Carlo Tree Search (MCTS) gradually explores the state space and approaches an optimal or suboptimal solution.

Double Q-Learning: The Double Q-learning [50] algorithm is a variant of the Q-learning algorithm which seeks to mitigate the problem of overestimation bias observed in conventional Q-learning. The overestimation bias arises in the Q-learning technique when state-action pairs are overvalued during the process of learning. The aforementioned bias has the potential to result in suboptimal policy selection and impede the convergence of Q-values. The Double Q-learning algorithm mitigates the problem of overestimation bias encountered by conventional Q-learning through the implementation of two distinct sets of action-value functions.

The standard Q-learning maintains a single set of Q-values and updates the action-value using the following equation:

$$Q(s,e) \leftarrow Q(s,e) + \alpha[R(e,s,s') + \gamma \cdot max_{e \in E_{s'}} Q(s',e^*) - Q(s,e)] \tag{7.1}$$

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

Double Q-learning maintains two sets of action-value functions, typically denoted as Q1 and Q2. These two sets are employed to decouple the processes of action selection and evaluation by utilizing one set of Q-values for the selection of optimal action, and the other set for evaluating the value of that action. The determination of the Q-value set to be utilized for value updates is typically determined by a switching mechanism. The switching mechanism may be predicated on a predetermined schedule or a random selection. The update equation for Double Q-learning is as follows:

$$Q1(s,e) \leftarrow Q1(s,e) + \alpha[R(e,s,s') + \gamma \cdot max_{e \in E_{s'}} Q2(s',e^*) - Q1(s,e)] \tag{7.2}$$

$$Q2(s,e) \leftarrow Q2(s,e) + \alpha[R(e,s,s') + \gamma \cdot max_{e \in E_{s'}} Q1(s',e^*) - Q2(s,e)] \tag{7.3}$$

where $Q1(s,e)$ represents the Q-value for executing event e in state s using Q1 set of values and $Q2(s,e)$ represents the Q-value for executing event e in state s using Q2 set of values.

By uncoupling the process of action selection and evaluation, Double Q-learning effectively diminishes the overestimation bias that is commonly observed in the customary Q-learning approach.

7.3. Alternative Techniques for Test Suite Prioritization

In an effort to augment the effectiveness of prioritization procedures, alternative methodologies will be investigated. These strategies include using machine learning tech-

niques, genetic algorithms, or neural networks to help in prioritization. The incorporation of these sophisticated techniques is anticipated to refine and optimize the process of prioritization, consequently yielding enhanced efficiency and effectiveness of the test suites.

7.4. Application in Other Domains

The current research examines the hybrid methodology that combines reinforcement learning-based test generation and prioritization optimization specifically applied within the context of Android applications running on mobile devices. Subsequent investigations may examine other applications in other domains such as IoT (Internet of Things), smart watches, autonomous vehicles, and other event-driven systems.

REFERENCES

[1] *Android UI/Application Exerciser Monkey*, https://developer.android.com/studio/test/monkey.html, Accessed: 2023-05-05.

[2] *The app attention index 2021: Who takes the rap for the app?*, https://www.appdynamics.com/c/dam/r/appdynamics/Gated-Assets/analyst-reports/AppDynamics-App-Attention-Index-2021.pdf, 2021 AppDynamics LLC, Accessed: 05-19-2023.

[3] *Appium: Mobile app automation made awesome*, http://appium.io/,Accessed: 05-21-2023.

[4] *Emma*, http://emma.sourceforge.net/,Accessed:05-21-2023.

[5] *F-droid - free and open source android app repository*, https://f-droid.org/,Accessed:05-21-2023.

[6] *Flutter (2021). beautiful native apps in record time.*, https://flutter.dev/, Accessed: 05-19-2023.

[7] *Iconic-cross platform mobile app development framework.*, https://ionicframework.com/, Accessed: 05-19-2023.

[8] *Jacoco java code coverage library*, https://www.eclemma.org/jacoco/index.html, Accessed: 05-24-2023.

[9] *React native-a framework for building native apps using react.*, https://reactnative.dev/, Accessed: 05-19-2023.

[10] *UIAutomator*, https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html,Accessed:05-21-2023.

[11] David Adamo, Renée Bryce, and Tariq M. King, *Randomized event sequence generation strategies for automated testing of android apps*, Information Technology - New Generations (Shahram Latifi, ed.), Springer International Publishing, 2018, https://doi.org/10.1007/978-3-319-54978-1_72, pp. 571–578.

[12] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce, *Reinforce-*

ment learning for android gui testing, Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018, Association for Computing Machinery, 2018, https://doi.org/10.1145/3278186.3278187, pp. 2 – 8.

[13] David Adamo, Dmitry Nurmuradov, Shraddha Piparia, and Renée Bryce, *Combinatorial-based event sequence testing of android applications*, Information and Software Technology 99 (2018), 98–117, https://doi.org/10.1016/j.infsof.2018.03.007.

[14] Tanwir Ahmad, Adnan Ashraf, Dragos Truscan, and Ivan Porres, *Exploratory performance testing using reinforcement learning*, 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2019, https://doi.org/10.1109/SEAA.2019.00032, pp. 156–163.

[15] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer, *Improving dynamic analysis of android apps using hybrid test input generation*, 2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security), 2017, https://doi.org/10.1109/CyberSecPODS.2017.8074845, pp. 1–8.

[16] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, and Porfirio Tramontana, *Agrippin: A novel search based testing technique for android applications*, Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015, Association for Computing Machinery, 2015, https://doi.org/10.1145/2804345.2804348, p. 5–12.

[17] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon, *Exploiting the saturation effect in automatic random testing of android applications*, 2015 2nd ACM International Conference on Mobile Software Engineering and Systems, 2015, https://doi.org/10.1109/MobileSoft.2015.11, pp. 33–43.

[18] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon, *Using gui ripping for automated testing of android*

applications, Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (New York, NY, USA), ASE '12, Association for Computing Machinery, 2012, https://doi.org/10.1145/2351676.2351717, p. 258–261.

[19] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon, *Mobiguitar: Automated model-based testing of mobile apps*, IEEE Software 32 (2015), no. 5, 53 – 59, https://doi.org/10.1109/MS.2014.55.

[20] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan, *An introduction to mcmc for machine learning*, Machine learning 50 (2003), 5–43, https://doi.org/10.1023/A:1020281327116.

[21] Android Developer Guides, *Application fundamentals*, https://developer.android.com/guide/components/fundamentals, Accessed: 05-19-2023.

[22] Andrea Arcuri and Lionel Briand, *A practical guide for using statistical tests to assess randomized algorithms in software engineering*, Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, Association for Computing Machinery, 2011, https://doi.org/10.1145/1985793.1985795, p. 1–10.

[23] Michael Auer, Felix Adler, and Gordon Fraser, *Improving search-based android test generation using surrogate models*, Search-Based Software Engineering (Cham) (Mike Papadakis and Silvia Regina Vergilio, eds.), Springer International Publishing, 2022, https://doi.org/10.1007/978-3-031-21251-2_4, pp. 51 – 66.

[24] Tanzirul Azim and Iulian Neamtiu, *Targeted and depth-first exploration for systematic testing of android apps*, SIGPLAN Not. 48 (2013), no. 10, 641–660, https://doi.org/10.1145/2544173.2509549.

[25] Young-Min Baek and Doo-Hwan Bae, *Automated model-based android gui testing using multi-level gui comparison criteria*, ASE '16, Association for Computing Machinery, 2016, https://doi.org/10.1145/2970276.2970313, pp. 238 – 249.

[26] Anu Bajaj and Om Prakash Sangwan, *A systematic literature review of test case prioritization using genetic algorithms*, IEEE Access 7 (2019), 126355 – 126375, https://doi.org/10.1109/ACCESS.2019.2938260.

[27] Renée C. Bryce and Atif M. Memon, *Test suite prioritization by interaction coverage*, DOSTA '07, Association for Computing Machinery, 2007, https://doi.org/10.1145/1294921.1294922, pp. 1 – 7.

[28] Renee C. Bryce, Sreedevi Sampath, and Atif M. Memon, *Developing a single model and test prioritization strategies for event-driven software*, IEEE Transactions on Software Engineering 37 (2011), no. 1, 48 – 64, https://doi.org/10.1109/TSE.2010.12.

[29] Renée C. Bryce, Sreedevi Sampath, Jan Bækgaard Pedersen, and Schuyler Manchester, *Test suite prioritization by cost-based combinatorial interaction coverage*, International Journal of System Assurance Engineering and Management 2 (2011), 126 – 134, https://doi.org/10.1007/s13198-011-0067-4.

[30] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh, *Deep reinforcement fuzzing*, 2018, https://doi.org/10.48550/ARXIV.1801.04589.

[31] Yuzhong Cao, Guoquan Wu, Wei Chen, and Jun Wei, *Crawldroid: Effective model-based gui testing of android apps*, Proceedings of the Tenth Asia-Pacific Symposium on Internetware, Internetware '18, Association for Computing Machinery, 2018, https://doi.org/10.1145/3275219.3275238, pp. 1 – 6.

[32] Wontae Choi, George Necula, and Koushik Sen, *Guided gui testing of android apps with minimal restart and approximate learning*, Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications, OOPSLA '13, Association for Computing Machinery, 2013, urlhttps://doi.org/10.1145/2509136.2509552, p. 623–640.

[33] S. Choudhary, A. Gorla, and A. Orso, *Automated test input generation for android: Are we there yet? (e)*, 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Computer Society, 2015, https://doi.org/10.1109/ASE.2015.89, pp. 429–440.

[34] Eliane Collins, Arilo Neto, Auri Vincenzi, and José Maldonado, *Deep reinforcement learning based android application gui testing*, Proceedings of the XXXV Brazilian

Symposium on Software Engineering, SBES '21, Association for Computing Machinery, 2021, https://doi.org/10.1145/3474624.3474634, p. 186–194.

[35] Céline Craye, David Filliat, and Jean-François Goudou, *Rl-iac: An exploration policy for online saliency learning on an autonomous mobile robot*, 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2016, https://doi.org/10.1109/IROS.2016.7759716, pp. 4877–4884.

[36] Umakanta Dash and Arup Abhinna Acharya, *A systematic review of test case prioritization approaches*, Proceedings of International Conference on Advanced Computing Applications (Jyotsna Kumar Mandal, Rajkumar Buyya, and Debashis De, eds.), Springer Singapore, 2022, https://doi.org/10.1007/978-981-16-5207-3_55, pp. 653 – 666.

[37] Hyunsook Do, Gregg Rothermel, and Alex Kinneer, *Prioritizing junit test cases: An empirical assessment and cost-benefits analysis*, Empirical Software Engineering 11 (2006), 33 – 70, https://doi.org/10.1007/s10664-006-5965-8.

[38] S. Elbaum, A. Malishevsky, and G. Rothermel, *Incorporating varying test costs and fault severities into test case prioritization*, Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, 2001, https://doi.org/10.1109/ICSE.2001.919106, pp. 329–338.

[39] Marcelo Medeiros Eler, Jose Miguel Rojas, Yan Ge, and Gordon Fraser, *Automated accessibility testing of mobile apps*, 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018, https://doi.org/10.1109/ICST.2018.00021, pp. 116–126.

[40] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth, *Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones*, ACM Trans. Comput. Syst. 32 (2014), no. 2, https://doi.org/10.1145/2619091.

[41] Ana Rosario Espada, María del Mar Gallardo, Alberto Salmerón, and Pedro Merino, *Using model checking to generate test cases for android applications*, Electronic Pro-

ceedings in Theoretical Computer Science 180 (2015), 7–21, https://doi.org/10.4204/eptcs.180.1.

[42] Anna I. Esparcia-Alcázar, Francisco Almenar, Mirella Martínez, Urko Rueda, and Tanja E. J. Vos, *Q-learning strategies for action selection in the testar automated testing tool*, Proceedings of the 6th International Conferenrence on Metaheuristics and Nature Inspired Computing (META 2016), 2016, pp. 130 – 137.

[43] Chunrong Fang, Zhenyu Chen, and Baowen Xu, *Comparing logic coverage criteria on test case prioritization*, Science China Information Sciences 55 (2012), 2826–2840.

[44] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li, *S¡span class="smallcaps smallercapital"¿eads¡/span¿: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning*, ACM Trans. Softw. Eng. Methodol. 30 (2021), no. 1, https://doi.org/10.1145/3379345.

[45] Pooja Goyal, Md Khorrom Khan, Christian Steil, Sarah M. Martel, and Renee Bryce, *Smartphone context event sequence prediction with poermh and tke-rules algorithms*, 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC), 2023, https://doi.org/10.1109/CCWC57344.2023.10099166, pp. 0827–0834.

[46] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü, *Aimdroid: Activity-insulated multi-level automated testing for android applications*, 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, https://doi.org/10.1109/ICSME.2017.72, pp. 103–114.

[47] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su, *Practical gui testing of android applications via model abstraction and refinement*, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, https://doi.org/10.1109/ICSE.2019.00042, pp. 269–280.

[48] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie, *To be opti-*

mal or not in test-case prioritization, IEEE Transactions on Software Engineering 42 (2016), no. 5, 490 – 505, https://doi.org/10.1109/TSE.2015.2496939.

[49] L. V. Haoyin, *Automatic android application gui testing—a random walk approach*, 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), 2017, https://doi.org/10.1109/WiSPNET.2017.8299722, pp. 72–76.

[50] Hado Hasselt, *Double q-learning*, Advances in neural information processing systems 23 (2010).

[51] Charitha Hettiarachchi, Hyunsook Do, and Byoungju Choi, *Risk-based test case prioritization using a fuzzy expert system*, Information and Software Technology 69 (2016), 1–15, 10.1016/j.infsof.2015.08.008.

[52] Cuixiong Hu and Iulian Neamtiu, *Automating GUI testing for Android applications*, Proceedings of the 6th International Workshop on Automation of Software Test, ACM, 2011, pp. 77–83.

[53] Gang Hu, Linjie Zhu, and Junfeng Yang, *Appflow: Using machine learning to synthesize robust, reusable ui tests*, Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, Association for Computing Machinery, 2018, https://doi.org/10.1145/3236024.3236055, p. 269–282.

[54] Rubing Huang, Quanjun Zhang, Dave Towey, Weifeng Sun, and Jinfu Chen, *Regression test case prioritization by code combinations coverage*, Journal of Systems and Software 169 (2020), 110712, https://doi.org/10.1016/j.jss.2020.110712.

[55] Rubing Huang, Weiwen Zong, Jinfu Chen, Dave Towey, Yunan Zhou, and Deng Chen, *Prioritizing interaction test suites using repeated base choice coverage*, 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), vol. 1, 2016, https://doi.org/10.1109/COMPSAC.2016.167, pp. 174–184.

[56] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang, *A history-based cost-cognizant test*

case prioritization technique in regression testing, Journal of Systems and Software 85 (2012), no. 3, 626 – 637, https://doi.org/10.1016/j.jss.2011.09.063.

[57] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek, *Search-based energy testing of android*, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, https://doi.org/10.1109/ICSE.2019.00115, pp. 1119–1130.

[58] Ajay Kumar Jha, Sooyong Jeong, and Woo Jin Lee, *Value-deterministic search-based replay for android multithreaded applications*, Proceedings of the 2013 Research in Adaptive and Convergent Systems (New York, NY, USA), RACS '13, Association for Computing Machinery, 2013, https://doi.org/10.1145/2513228.2513279, p. 381–386.

[59] Bo Jiang, Yaoyue Zhang, Wing Kwong Chan, and Zhenyu Zhang, *A systematic study on factors impacting gui traversal-based test case generation techniques for android applications*, IEEE Transactions on Reliability 68 (2019), no. 3, 913–926, https://doi.org/10.1109/TR.2019.2928459.

[60] J.A. Jones and M.J. Harrold, *Test-suite reduction and prioritization for modified condition/decision coverage*, Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, 2001, https://doi.org/10.1109/ICSM.2001.972715, pp. 92–101.

[61] Md Khorrom Khan and Renee Bryce, *Android gui test generation with sarsa*, 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), 2022, https://doi.org/10.1109/CCWC54503.2022.9720807, pp. 0487 – 0493.

[62] Md Khorrom Khan, Ryan Michaels, Farhan Rahman Arnob, and Renée Bryce, *Prioritization techniques for android test suites generated by a reinforcement learning algorithm*, (2021), Available at SSRN: https://ssrn.com/abstract=4450321 or http://dx.doi.org/10.2139/ssrn.4450321.

[63] Md Khorrom Khan, Ryan Michaels, Dylan Williams, Benjamin Dinal, Beril Gurkas, Austin Luloh, and Renee Bryce, *Post prioritization techniques to improve code coverage*

for sarsa generated test cases, 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC), 2023, https://doi.org/10.1109/CCWC57344.2023.10099120, pp. 1029 – 1035.

[64] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N. A. Jawawi, Muhammad Luqman Mohd Shafie, Wan Mohd Nasir Wan-Kadir, Haza Nuzly Abdull Hamed, and Muhammad Dhiauddin Mohamed Suffian, *Trend application of machine learning in test case prioritization: A review on techniques*, IEEE Access 9 (2021), 166262 – 166282, https://doi.org/10.1109/ACCESS.2021.3135508.

[65] Junhwi Kim, Minhyuk Kwon, and Shin Yoo, *Generating test input with deep reinforcement learning*, Proceedings of the 11th International Workshop on Search-Based Software Testing, SBST '18, Association for Computing Machinery, 2018, https://doi.org/10.1145/3194718.3194720, p. 51–58.

[66] Pavneet Singh Kochhar, Ferdian Thung, and David Lo, *Code coverage and test suite effectiveness: Empirical study with real bugs in large systems*, 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, https://doi.org/10.1109/SANER.2015.7081877, pp. 560 – 564.

[67] Patipat Konsaard and Lachana Ramingwong, *Total coverage based regression test case prioritization using genetic algorithm*, 2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2015, https://doi.org/10.1109/ECTICon.2015.7207103, pp. 1–6.

[68] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi, *Pyse: Automatic worst-case test generation by reinforcement learning*, 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, https://doi.org/10.1109/ICST.2019.00023, pp. 136–147.

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

[70] Herb Krasner, *The cost of poor software quality in the us: A 2020 report*, The Consortium for Information Software Quality™ (CISQ™), https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf, Accessed: 05-19-2023.

[71] Alexandr Kuznetsov, Yehor Yeromin, Oleksiy Shapoval, Kyrylo Chernov, Mariia Popova, and Kostyantyn Serdukov, *Automated software vulnerability testing using deep learning methods*, 2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering (UKRCON), 2019, https://doi.org/10.1109/UKRCON.2019.8879997, pp. 837–841.

[72] Duling Lai and Julia Rubin, *Goal-driven exploration for android applications*, 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, https://doi.org/10.1109/ASE.2019.00021, pp. 115–127.

[73] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen, *Droidbot: a lightweight ui-guided test input generator for android*, 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017, https://doi.org/10.1109/ICSE-C.2017.8, pp. 23–26.

[74] Zheng Li, Mark Harman, and Robert M. Hierons, *Search algorithms for regression test case prioritization*, IEEE Transactions on Software Engineering 33 (2007), no. 4, 225 – 237, https://doi.org/10.1109/TSE.2007.38.

[75] Yang Liu, Huaping Liu, and Bowen Wang, *Autonomous exploration for mobile robot using q-learning*, 2017 2nd International Conference on Advanced Robotics and Mechatronics (ICARM), 2017, https://doi.org/10.1109/ICARM.2017.8273233, pp. 614–619.

[76] Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li, *Preference-wise testing for android applications*, Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, Association for Computing Machinery, 2019, https://doi.org/10.1145/3338906.3338980, p. 268–278.

[77] Aravind Machiry, Rohan Tahiliani, and Mayur Naik, *Dynodroid: An input generation*

system for android apps, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, Association for Computing Machinery, 2013, https://doi.org/10.1145/2491411.2491450, p. 224–234.

[78] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek, *Evodroid: Segmented evolutionary testing of android apps*, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Association for Computing Machinery, 2014, https://doi.org/10.1145/2635868.2635896, p. 599–609.

[79] A.G. Malishevsky, G. Rothermel, and S. Elbaum, *Modeling the cost-benefits tradeoffs for regression testing techniques*, International Conference on Software Maintenance, 2002. Proceedings., 2002, https://doi.org/10.1109/ICSM.2002.1167767, pp. 204–213.

[80] Ke Mao, Mark Harman, and Yue Jia, *Sapienz: Multi-objective automated testing for android applications*, Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Association for Computing Machinery, 2016, https://doi.org/10.1145/2931037.2931054, p. 94–105.

[81] Alessandro Marchetto, Md. Mahfuzul Islam, Waseem Asghar, Angelo Susi, and Giuseppe Scanniello, *A multi-objective technique to prioritize test cases*, IEEE Trans. Softw. Eng. 42 (2016), no. 10, 918 – 940, https://doi.org/10.1109/TSE.2015.2510633.

[82] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro, *Autoblacktest: a tool for automatic black-box testing*, 2011 33rd International Conference on Software Engineering (ICSE), 2011, https://doi.org/10.1145/1985793.1985979, pp. 1013–1015.

[83] Dusica Marijan, *Multi-perspective regression test prioritization for time-constrained environments*, 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015, https://doi.org/10.1109/QRS.2015.31, pp. 157 – 162.

[84] Ryan Michaels, Md Khorrom Khan, and Renée Bryce, *Mobile test suite generation via combinatorial sequences*, ITNG 2021 18th International Conference on Information

Technology-New Generations (Shahram Latifi, ed.), Springer International Publishing, 2021, https://doi.org/10.1007/978-3-030-70416-2_35, pp. 273–279.

[85] Ryan Michaels, Md Khorrom Khan, and Renee Bryce, *Test suite prioritization with element and event sequences for android applications*, 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC), 2021, https://doi.org/10.1109/CCWC51732.2021.9376143, pp. 1326 – 1332.

[86] Anil Mor, *Evaluate the effectiveness of test suite prioritization techniques using apfd metric*, IOSR Journal of Computer Engineering 16 (2014), 47–51.

[87] Rajendrani Mukherjee and K. Sridhar Patnaik, *A survey on different approaches for software test case prioritization*, Journal of King Saud University - Computer and Information Sciences 33 (2021), no. 9, 1041 – 1054, https://doi.org/10.1016/j.jksuci.2018.09.005.

[88] Liming Nie, Kabir Sulaiman Said, Lingfei Ma, Yaowen Zheng, and Yangyang Zhao, *A systematic mapping study for graphical user interface testing on mobile apps*, IET Software (2023), https://doi.org/10.1049/sfw2.12123.

[89] Tanzeem Bin Noor and Hadi Hemmati, *A similarity-based approach for test case prioritization using historical failure data*, 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), 2015, https://doi.org/10.1109/ISSRE.2015.7381799, pp. 58 – 68.

[90] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li, *Reinforcement learning based curiosity-driven testing of android applications*, Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, 2020, https://doi.org/10.1145/3395363.3397354, p. 153–164.

[91] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik, *Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing*, 2008 Second International Conference on Secure System Integration and Reliability Improvement, 2008, https://doi.org/10.1109/SSIRI.2008.52, pp. 39 – 46.

[92] Chao Peng, Zhao Zhang, Zhengwei Lv, and Ping Yang, *Mubot: Learning to test large-scale commercial android apps like a human*, 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022, https://doi.org/10.1109/ICSME55016.2022.00074, pp. 543–552.

[93] Petroc Taylor, *Population of internet users worldwide from 2012 to 2022, by operating system(in millions)*, https://www.statista.com/statistics/543185/worldwide-internet-connected-operating-system-population/, Statista 2023, Accessed: 05-19-2023.

[94] Shraddha Piparia, Md Khorrom Khan, and Renée Bryce, *Discovery of real world context event patterns for smartphone devices using conditional random fields*, ITNG 2021 18th International Conference on Information Technology-New Generations (Cham) (Shahram Latifi, ed.), Springer International Publishing, 2021, https://doi.org/10.1007/978-3-030-70416-2_29, pp. 221–227.

[95] Ju Qian and Di Zhou, *Prioritizing test cases for memory leaks in android applications*, Journal of Computer Science and Technology 31 (2016), 869 – 882, https://doi.org/10.1007/s11390-016-1670-2.

[96] R. Krishnamoorthi and S.A. Sahaaya Arul Mary, *Factor oriented requirement coverage based system test case prioritization of new and regression test cases*, Information and Software Technology 51 (2009), no. 4, 799 – 808, https://doi.org/10.1016/j.infsof.2008.08.007.

[97] Andrea Romdhana and Alessio Merlo, *Keynote: Ares: A deep reinforcement learning tool for black-box testing of android apps*, 2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops), 2021, https://doi.org/10.1109/PerComWorkshops51409.2021.9431072, pp. 173–173.

[98] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang, *Automation of android applications functional testing using machine learning activities classification*, Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILE-

Soft '18, Association for Computing Machinery, 2018, https://doi.org/10.1145/3197231.3197241, p. 122–132.

[99] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold, *Prioritizing test cases for regression testing*, IEEE Transactions on Software Engineering 27 (2001), no. 10, 929 – 948, https://doi.org/10.1109/32.962562.

[100] Konstantin Rubinov and Luciano Baresi, *What are we missing when testing our android apps?*, Computer 51 (2018), no. 4, 60–68, https://doi.org/10.1109/MC.2018.2141024.

[101] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek, *Patdroid: Permission-aware gui testing of android*, Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, 2017, https://doi.org/10.1145/3106237.3106250, p. 220 – 232.

[102] Yasmine Ibrahim Salem and Riham Hassan, *Requirement-based test case generation and prioritization*, 2010 International Computer Engineering Conference (ICENCO), 2010, https://doi.org/10.1109/ICENCO.2010.5720443, pp. 152 – 157.

[103] Sreedevi Sampath, Renée Bryce, and Atif M. Memon, *A uniform representation of hybrid criteria for regression testing*, IEEE Transactions on Software Engineering 39 (2013), no. 10, 1326 – 1344, https://doi.org/10.1109/TSE.2013.16.

[104] Preeti Satish, Peri Nikhil, and Krishnan Rangarajan, *A test prioritization algorithm that cares for "don't care" values and higher order combinatorial coverage*, SIGSOFT Softw. Eng. Notes 42 (2018), no. 4, 1–9, https://doi.org/10.1145/3149485.3149510.

[105] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley, *All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)*, 2010 IEEE Symposium on Security and Privacy, 2010, https://doi.org/10.1109/SP.2010.26, pp. 317–331.

[106] Wei Song, Xiangxing Qian, and Jeff Huang, *Ehbdroid: Beyond gui testing for android applications*, 2017 32nd IEEE/ACM International Conference on Automated

Software Engineering (ASE), IEEE, 2017, https://doi.org/10.1109/ASE.2017.8115615, pp. 27 – 37.

[107] Hema Srikanth, Myra B. Cohen, and Xiao Qu, *Reducing field failures in system configurable software: Cost-based prioritization*, 2009 20th International Symposium on Software Reliability Engineering, 2009, https://doi.org/10.1109/ISSRE.2009.26, pp. 61–70.

[108] Praveen Ranjan Srivastva, Krishan Kumar, and G Raghurama, *Test case prioritization based on requirements and risk factors*, SIGSOFT Softw. Eng. Notes 33 (2008), no. 4, https://doi.org/10.1145/1384139.1384146.

[109] Heiko Stallbaum, Andreas Metzger, and Klaus Pohl, *An automated technique for risk-based test case generation and prioritization*, Proceedings of the 3rd International Workshop on Automation of Software Test, AST '08, Association for Computing Machinery, 2008, https://doi.org/10.1145/1370042.1370057, p. 67–70.

[110] Ting Su, *Fsmdroid: Guided gui testing of android apps*, 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), Association for Computing Machinery, 2016, https://doi.org/10.1145/2889160.2891043, pp. 689–691.

[111] Ting Su, 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*, Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, 2017, https://doi.org/10.1145/3106237.3106298, p. 245–256.

[112] Richard S Sutton and Andrew G Barto, *Reinforcement learning: An introduction*, MIT press, 2018.

[113] Petroc Taylor, *Mobile operating systems' market share worldwide from 1st quarter 2009 to 4th quarter 2022*, https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, Statista 2023, Accessed: 05-19-2023.

[114] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino, *Automated functional testing of mobile applications: a systematic mapping study*, Software Quality Journal 27 (2019), 149–201, https://doi.org/10.1007/s11219-018-9418-6.

[115] Martijn van Otterlo and Marco Wiering, *Reinforcement learning and markov decision processes*, pp. 3–42, Springer Berlin Heidelberg, 2012, https://doi.org//10.1007/978-3-642-27645-3_1.

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

[117] Muhammad Waqar, Imran, Muhammad Atif Zaman, Muhammad Muzammal, and Jungsuk Kim, *Test suite prioritization based on optimization approach using reinforcement learning*, Applied Sciences 12 (2022), no. 13, article no: 6772, https:doi.org/10.3390/app12136772.

[118] Hsiang-Lin Wen, Chia-Hui Lin, Tzong-Han Hsieh, and Cheng-Zen Yang, *Pats: A parallel gui testing framework for android applications*, 2015 IEEE 39th Annual Computer Software and Applications Conference, vol. 2, 2015, https://doi.org/10.1109/COMPSAC.2015.80, pp. 210–215.

[119] Thomas D. White, Gordon Fraser, and Guy J. Brown, *Improving random gui testing with image-based widget detection*, Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Association for Computing Machinery, 2019, https://doi.org/10.1145/3293882.3330551, p. 307–317.

[120] Wen-Chieh Wu and Shih-Hao Hung, *Droiddolphin: A dynamic android malware detection framework using big data and machine learning*, Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, RACS '14, Associa-

tion for Computing Machinery, 2014, https://doi.org/10.1145/2663761.2664223, p. 247–252.

[121] Jiwei Yan, Linjie Pan, Yaqi Li, Jun Yan, and Jian Zhang, *Land: A user-friendly and customizable test generation tool for android apps*, Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (New York, NY, USA), ISSTA 2018, Association for Computing Machinery, 2018, https://doi.org/10.1145/3213846.3229500, p. 360–363.

[122] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang, *Widget-sensitive and back-stack-aware gui exploration for testing android apps*, 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, https://doi.org/10.1109/QRS.2017.14, pp. 42–53.

[123] Wei Yang, Mukul R. Prasad, and Tao Xie, *A grey-box approach for automated gui-model generation of mobile applications*, Fundamental Approaches to Software Engineering (Vittorio Cortellessa and Dániel Varró, eds.), Springer Berlin Heidelberg, 2013, https://doi.org/10.1007/978-3-642-37057-1_19, pp. 250–265.

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

[125] Miso Yoon, Eunyoung Lee, Mikyoung Song, Byoungju Choi, et al., *A test case prioritization through correlation of requirement and risk*, Journal of Software Engineering and Applications 5 (2012), no. 10, 823, 10.4236/jsea.2012.510095.

[126] Naoto Yoshida, Eiji Uchibe, and Kenji Doya, *Reinforcement learning with state-dependent discount factor*, 2013 IEEE Third Joint International Conference on Development and Learning and Epigenetic Robotics (ICDL), 2013, https://doi.org/10.1109/DevLrn.2013.6652533, pp. 1–6.

[127] Linbin Yu, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn, *Acts: A combinatorial test generation tool*, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, https://doi.org/10.1109/ICST.2013.52, pp. 370–375.

[128] Samer Zein, Norsaremah Salleh, and John Grundy, *A systematic mapping study of mobile application testing techniques*, Journal of Systems and Software 117 (2016), 334–356, https://doi.org/10.1016/j.jss.2016.03.065.

[129] Xiaofang Zhang, Changhai Nie, Baowen Xu, and Bo Qu, *Test case prioritization based on varying testing requirement priorities and test case costs*, Seventh International Conference on Quality Software (QSIC 2007), 2007, https://doi.org/10.1109/QSIC.2007.4385476, pp. 15 – 24.

[130] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci, *Towards black box testing of android apps*, 2015 10th International Conference on Availability, Reliability and Security, 2015, https://doi.org/10.1109/ARES.2015.70, pp. 501–510.

[131] Haowen Zhu, Xiaojun Ye, Xiaojun Zhang, and Ke Shen, *A context-aware approach for dynamic gui testing of android applications*, 2015 IEEE 39th Annual Computer Software and Applications Conference, vol. 2, 2015, https://doi.org/10.1109/COMPSAC.2015.77, pp. 248–253.