Android Game Testing using Reinforcement Learning

Suhani Khurana

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Na Meng, Chair
Bimal Viswanath
Muhammad A. Gulzar

May 3, 2023
Blacksburg, Virginia

Keywords: Android, Testing, Reinforcement Learning, Software Engineering

Android Game Testing using Reinforcement Learning

Suhani Khurana

(ABSTRACT)

Android is the most popular operating system and occupies close to 70% of the market share. With the growth in the usage of Android OS, the number of games also increased and the Android play store has over 500,000 games. Testing of Android games is done either manually or through some of the existing tools which automate some part of this testing. Manual testing requires a great deal of effort and can be expensive to afford. The existing tools which automate testing do not make use of any domain knowledge. This can cause the testing to be ineffective as the game may involve complex strategies, intricate details and widgets, etc. Existing tools like Android Monkey and Time Machine generate random Android events, including gestures like touch, swipe, clicks and other system level events across the application. Some deep learning methods like Wuji was only created for combat type games. These limitations make it imperative to create a testing paradigm which uses domain knowledge as well as is easy to use by a developer who doesn't have any machine or deep learning knowledge.

In this work, we develop a tool called DRAG - Deep Reinforcement learning based Android Gamer - which leverages Reinforcement Learning to learn the requisite domain knowledge and play the game in a fashion like a human would. DRAG uses a unified Reinforcement Learning agent and a Unified Reinforcement Learning environment. It only customises the action space for each game. This geenralisation is done in the following ways- 1) Record an 8 minute demo video of the game and capture the underlying Android action log. 2) Analyze the recorded video and the action log to generate an action space for the Reinforcement

Learning Agent. The unified RL agent is trained by providing it the score and coverage as reward and screenshots of the game as observed states. We chose a set of 19 different open sourced games for evaluation of the created tool. These games differ in the action set required by each of them - some require tapping icons, some require swiping in random directions, and some require more complex actions which are a combination of different gestures.

The evaluation of our tool outperformed state-of-the-art TimeMachine for all the 19 games, and outperformed Monkey in 16 of the 19 games. This strengthens the fact that Deep Reinforcement Learning can be used to test Android games and can provide better results than tools which make no use of any domain knowledge.

Android Game Testing using Reinforcement Learning

Suhani Khurana

(GENERAL AUDIENCE ABSTRACT)

The popularity of the Android operating system has led to a significant increase in the number of available Android games, with over 500,000 games on the Android Play Store alone. However, ensuring the quality and functionality of these games has become a challenge. Traditional testing methods involve either time-consuming manual testing or the use of existing tools that lack the necessary domain knowledge to handle complex game mechanics effectively.

To overcome these limitations, we propose a solution called DRAG : the Deep Reinforcement Learning based Android Gamer. Our tool utilizes Reinforcement Learning (RL) to acquire the domain knowledge needed to play Android games in a manner similar to human players. Unlike other tools, DRAG incorporates a unified RL agent and environment that can be customized for each specific game.

The process of customizing the action space involves two main steps. First, we record an 8-minute demonstration video of the game while capturing the underlying Android action log. Then, we analyze the video and action log to generate a tailored action space for the game. The unified RL agent is trained using rewards based on the game's score and coverage, while observed screenshots of the game serve as input states.

We evaluated DRAG using a diverse set of 19 open-source games, each requiring different actions such as tapping icons, swiping in random directions, or complex combinations of gestures. Our results demonstrate that DRAG outperforms state-of-the-art tools like

TimeMachine in all 19 games and outperforms Monkey in 16 of the 19 games. These findings highlight the effectiveness of Deep Reinforcement Learning for testing Android games and its ability to deliver better results compared to tools lacking domain knowledge.

Our work introduces a new testing approach that combines RL and domain knowledge, providing a user-friendly solution for developers without extensive machine or deep learning expertise. By automating game testing to replicate human gameplay, DRAG offers the potential for more efficient and effective quality assurance in the Android gaming ecosystem.

Dedication

To my parents Manju and Ram Khurana and my sister Yukti

Who motivated me to never give up, and keep striving to become better.

Acknowledgments

It has been quite a journey transitioning from corporate world back to school. I came to Virginia Tech in 2021, after having worked for 5 years as a software developer. I'm grateful for this opportunity. Dr Na Meng has been pivotal in this journey, having been there at every step to support this research, and providing encouragement even in my job search. This would not have been possible without her. I would also like to thank Dr. Vishwanath and Dr. Gulzar for serving on my committee. This research has been shaped into the way it is with support from Dr. Liu from Brandeis University and his student Yi-Zhe Hong for their support in computer vision related tasks. Dr Eldardiry and her student Vasanth Reddy helped me with many questions regarding Reinforcement Learning. I'm thankful to each of you for all your help and advise.

This research is not just a work of the technical tasks but also all the emotional support during this long journey of two years. I would like to thank my parents and my sister for being my support system and providing encouragement. My boyfriend Pronnoy has always inspired me to work hard and helped me with all his experiences as a graduate student - inspiring me to be a better version of myself everyday. This acknowledgement is incomplete if I don't extend my gratitude to my friends - Nandita, Pratijay, Ameya, Himanshu and Varun who have literally been my family here in a foreign land - supporting me through every up and down, and being there to celebrate every small achievement. This work would not have been what it is without any of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android OS has occupied 71% market share in 2023 [16] and has gained popularity. This also implies the number of applications and games on the Android Playstore keep increasing with the increasing number of users. In 4th quarter of 2022, there were nearly 490k Android games on the Google Playstore [27]. In addition to Google Playstore, there are a number of other playstores available as well which would further add more Android games to the pool. The increase in the number of games doesn't necessarily mean a proportional increase in the tools required to ensure quality. This can lead to games with bugs entering the playstore, and hence could mean losses both financial and reputational for the developers.

There are two possible ways in which Android applications (including games) can be tested - automated and manual. Due to the intricate detailing in the graphical user interface (GUI) and presence of different functional icons, a lot of software companies still rely on and prefer manual testing for such applications [36]. In addition, there are many scenarios in games which can only be uncovered after certain missions are achieved, in turn making it rely heavily on manual testing. But this manual testing can be taxing, is prone to human errors and is time and cost inefficient.

Automated testing has gained popularity and is being implemented more these days. Due to this, there have been extensive research on the same. Different techniques are being explored for automating Android testing. Some existing works rely on search-based techniques to explore game space [42], generating random actions [20, 34], fuzzing techniques [38, 49], etc.

But these techniques don't leverage any domain knowledge or game playing logic, but rather rely on random events or already existing test cases to explore the game further.

Machine learning, specifically Deep Learning, techniques are now reaching all walks of life. In recent years, these deep learning techniques have been explored for testing Android Games as well. Wuji [50] uses Deep reinforcement learning for testing combat based games, AlphaGO by Deep Mind for beating opponents at GO board game [45], DinoDroid which uses Deep-Q networks for GUI testing of Android applications [48]. But these approaches are either catered to only a few games or test just GUI without leveraging any game playing logic.

To overcome the challenges of the existing tools which make use of Deep Learning techniques, and make it more generalizable and applicable to newer games, we create a tool called DRAG (Deep Reinforcement learning based Android Gamer). While developing this tool, we faced various challenges:

1. Deep Reinforcement Learning needs an action space. The initial idea was to provide a generic action space for all games (which includes touch, swipe, long touch, etc) and let the agent learn the requisite actions for a game during training. But with approach, the action space becomes really large and sparse where only a very few actions would fetch rewards.

2. We decided that reward for the agent should include the game score (wherever available). But this is only avaiable from inside the game code, and to send to the DRL agent could mean modifications to the source code.

DRAG consists of a unified Reinforcmeent Learning agent, which has all the hyperparameters fixed for each game. It also consists of a unified Reinforcment Learning environment where the step methods are defined for all games. Both the unified reinforcement learning environment and agent don't need to be customised for each game.

For the action space, we need to customise the action space for each game. To do this, the developer records an eight minute demo video for a game. During this time period, the recording of the emulator is captured, along with the log trace.

After this, the captured log trace is analysed, mapped to each frame in the recorded video and the actions are analysed. In this step, vital information is added to the action log generated which includes information about the type of action (tap/swipe), duration of action, if the action was performed on a game icon or not, etc.

Finally, the Deep Reinforcement Learning Agent utilises the action space generated to test the game for a specific period of time, during which it goes through the exploration and exploitation cycles, adjusting the different model paramters. The DRL agent captures the screenshot of the game before and after each step, recognises the game score using PaddleOCR and utilises this as a reward in addition to the line coverage.

In this research, we aim to answer the following research questions:

RQ1 ***How does DRAG compare to other tools?*** For answering this question, we compare the effectiveness of 2 tools Monkey[20] and TimeMachine[34]. We run all the three tools for a duration of 30-minutes for the dataset of 19 games and capture the score and line coverage achieved by each tool.

RQ2 ***How does the effectiveness of DRAG change when standard GUI widgets are also tested?*** For answering this question, we compare add a 10 minute window to DRAG where it is allowed to take random actions and not game specific actions. For a fair comparison, we run the other two tools for a duration of 40-minutes as well. Similar to RQ1, we capture the score and line coverage.

RQ3 ***How does the effectiveness of DRAG change no icons are recognised by it?*** For answering this question, we created two variants of DRAG - one which recognises

menu icons and the other where no icon recognition is performed. Then, both varients are run on a subset of the games. This subset includes games where icon recognition was being performed initially. As above, the score and line coverage are captured for this experiment as well.

The different experiments performed pave various interesting observations.

- In comparison to Monkey and TimeMachine, DRAG is able to achieve a better line coverage for 16 of the 19 games in the dataset. It can achieve a better score for all the games where scores are available.
- When random actions are added, DRAG can achieve the same line coverage as Monkey can in the 40-minute window for the 3 games identified above. For the other 16 games, it is still able to achieve better line coverage and score (where available) than both the other tools.
- The effectiveness of DRAG is affected when icon recognition is not done. When menu icons only are recognised, it can still achieve better coverage than when no icons are recognised. This is because, for some games, the tool fails to even enter the game wihtout menu icons. For some games, the effectiveness doesn't get hampered a lot because we still supply game specific actions instead of random actions, which are just not performed on game icons but random locations.

For the rest of the thesis, we discuss some background in Chapter 2 and related works in Chapter 3. We then look at a motivating example in Chapter 4 and describe our approach in Chapter 5. The evaluation and answers to the research questions are discussed in Chapter 6 and Chapter 7. We conclude the study with threats to validity (Chapter 8), future works (Chapter 9) and conclusion (Chapter 10).

Chapter 2

Background

2.1 Android OS

Android Operating System is a mobile OS based on Linux kernel. It is an open source OS, and was developed back in 2007. Primarily developed by Google, most of the devices use this proprietary version created by Google. The other operating system prevalent in the market is iOS developed by Apple. iOS captures about 27% of the market while Android takes a whooping 71% [16]. Due to the large scale presence of Android OS devices, in this work we focus on Android games.

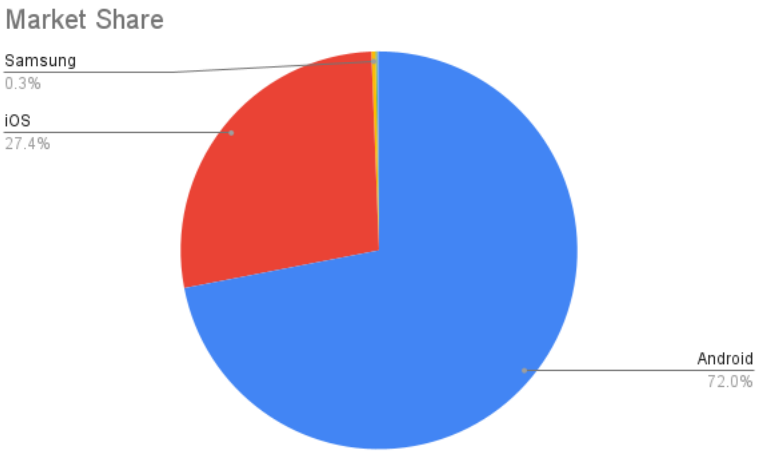

Figure 2.1: Market Share of different OS in Q1 2023 [16]

2.2 Software Testing

Software Testing refers to the practice of verification of a software system in terms of the intended functionalities and in order to find any bugs [33]. To ensure a quality product, a software must undergo a rigorous testing process. Generally, any software is either tested manually or via automated tool or a combination of both.

Automated testing can be performed in various ways - one way to perform automation testing is to include unit tests which are written by developers and test a unit of code in isolation. For instance JUnit framework created for Java can assist in writing test cases for Java programs [14]. Another way could be used by quality assurance personnel to translate business test cases into a test suite and run against the code everytime a build is triggered. This can be achieved by using Selenium framework [28].

Manual testing can be perfomed by a human, who understands the domain of the application, can foresee the scenarios that an application can run into, and can manually replicate these scenarios in order to determine the validity of the system. Both manual and automation testing verify the correctness, and also can uncover failure scenarios where the entire application may crash or is not robust.

2.2.1 Line Coverage

Line coverage is a code coverage metric that measures the percentage of lines of code that have been executed by a set of tests. In other words, it indicates the proportion of lines of code that have been covered by the tests, and can be used to assess the effectiveness and completeness of the tests.

To calculate line coverage, the number of lines of code that have been executed by the tests

is divided by the total number of lines of code in the program. This result is then expressed as a percentage, with a higher percentage indicating a higher degree of line coverage.

2.3 Reinforcement Learning

Reinforcement learning is a type of machine learning algorithm that is used to train artificial intelligence (AI) agents to learn and adapt to complex and dynamic environments. The goal of reinforcement learning is to enable the AI agent to learn and develop strategies and tactics for achieving a specific objective or goal, by receiving feedback and rewards based on its actions and decisions.

In reinforcement learning, the AI agent is placed in an environment, such as a game or a simulation, and it is given the ability to interact with the environment and to make decisions and take actions based on its observations and experiences. The AI agent is trained using a reinforcement learning algorithm, which adjusts the agent's behavior based on the feedback and rewards it receives from the environment. Over time, the AI agent learns to optimize its actions and decisions in order to maximize its rewards and achieve its objective.

2.3.1 Q-Learning

Q-learning refers to Quality Learning. Q-learning is a type of reinforcement learning algorithm that is used to train artificial intelligence (AI) agents to learn and adapt to complex and dynamic environments. The goal of Q-learning is to enable the AI agent to learn and develop strategies and tactics for achieving a specific objective or goal, by receiving feedback and rewards based on its actions and decisions. It is an off-policy method - that is separates out the acting policy from the learning policy.A Q-table is maintained with different actions

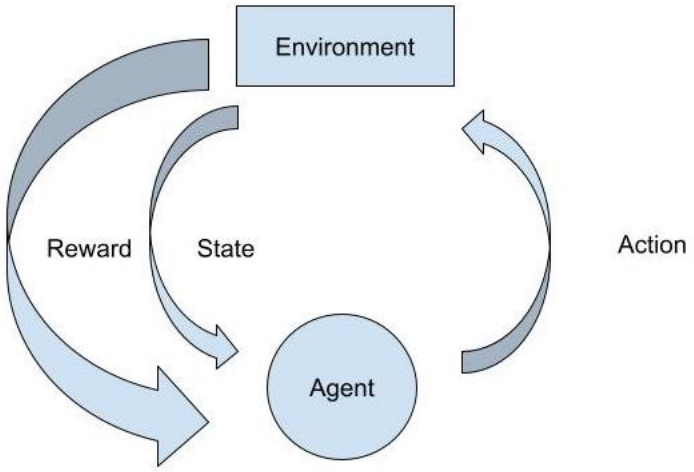

Figure 2.2: Reinforcement Learning

and their Q-values. These Q-values are calculated using the rewards and learning rate. Upon encountering a state, the action can be either be taken based on the Q-table or a new action is taken with a better Q-value, and the Q-table is updated.

Q-learning also has some disadvantages - since the Q-value is updated only once, for larger action spaces, it can take very long for it to get updated and there may be cases where some actions are not even updated.[39]

2.3.2 Deep Q-Learning

To enhance the powers of Q-learning, it is combined with Convoluted Neural Network. The experience replay which is a key concept of Q-learning provides stability to the value approximation of neural networks. DQN agents use the artificial neural network to extract many possible actions from the Q-tables and do it often, overcoming the select once shortcoming of the Q-learning.[43]

2.4 Tools and Libraries

2.4.1 OpenCV

OpenCV is an Open Source library used for computer vision tasks. It supports a host of CV related tasks like recognition of object in an image, stitching smaller images together to form a larger one, track movements in objects, etc. Although OpenCV itself is written in native C++, it provides interfaces for other languages like Python, Java and MATLAB, and also supports most of the popular operating systems including Windows, Linux, macOS, etc. [21]

2.4.2 Gym (now known as Gymnasium)

Gym (now known as Gymnasium) is an open-sourced library written in Python. It provides a standard interface for developing Reinforcement Learning environments and communication between the agent and the environment. In addition, it provides a set of some pre-written Gym environments for different applications like classic control, Atari games, etc. A user can also use Gym to create their own custom environment, which follows the standard API making it easy to integrate with any Reinforcement Learning agent. [12]

2.4.3 PaddleOCR

PaddleOCR is an open-source suite of tools and libraries for OCR and document analysis, developed by Baidu PaddlePaddle. It offers a range of tools and pre-trained models for tasks such as text recognition and document layout analysis, and supports multiple languages and fonts. Known for its performance and accuracy, PaddleOCR has been used in many research

projects and is a valuable resource for those working with OCR and document analysis. [35]

2.4.4 ADB

ADB is a command-line tool, included in the Android SDK development toolkit. It provides the ability for a development machine to run commands and debug an Android device or emulator. It makes use of a Unix shell to run commands on the device via a daemon (called adbd). ADB provides many capabilities like installing an application on device, performing actions on the device, capturing the trace log, recording screen, capturing screen, etc. It can be easily integrated with any other application to run different commands from the application on to the device or emulator. [1]

2.4.5 JaCoCo

The JaCoCo library is an open-source Java code coverage tool that enables developers to measure the effectiveness of their unit tests. By instrumenting code at run-time and recording its execution during unit tests, the JaCoCo library can determine which lines of code have been covered by tests and generate a report showing the overall coverage. It also offers a range of features, such as support for different code coverage metrics and integration with build tools and IDEs, and supports a variety of languages and platforms. JaCoCo is a valuable tool for developers looking to ensure their code is thoroughly tested and free from defects. [13]

Chapter 3

Review of Literature

To understand the previous works related to our line of work, we dive into the details of previous works done in Automated Testing of Android Apps and Games (Section 3.1) and Machine Learning based Android Game Testing (Section 3.2).

3.1 Automated Testing of Android Apps and Games

The most common and widely used form of Android Testing is random testing. Many applications for random testing have existed. Android Monkey [20] is used to generate random events like touch, tap, swipe, and other system level events. It aids in stress testing. But the major limitation of Android Monkey is that it has no knowledge of the system, and can only use standard widgets. For applications that do not have standard widgets, Monkey just generates random clicks which may or may not help in entering and using the application. Dong et al created another tool for random testing called TimeMachine [34].TimeMachine is a tool designed for testing Android apps that incorporates time-travel testing capabilities. It is a modified version of the Monkey testing tool and enables automated exploration of an app's various states by quickly performing random actions. It saves significant checkpoints during the testing process and allows for revisiting previously encountered states, thus maximizing the coverage of the app's functionality. TimeMachine specifically generates the sequence of actions that follow the most advanced and effective state of the app.

Another realm of testing techniques use search-based techniques to generate test cases.Mao et al created Sapienz [42], a tool which uses multi-objective exploratory techniques to test Android applications with the goal of maximising coverage and fault detection. It makes use of random fuzzing which requires test cases to be provided. Android apps don't always have test cases, and it requires effort to create these test cases in the first place. With any new changes, new test inputs may be required as well.

In Android, record and replay testing methods are utilized to automate the testing of app user interface (UI) interactions. The technique involves capturing user actions during interaction with the app, and then automatically replaying those actions to ensure that the app behaves correctly. This approach can save a lot of time and effort compared to manually testing each UI interaction [41]. Various tools are available in Android to facilitate record and replay testing. For instance, the Espresso testing framework [9] offers APIs for creating automated UI tests. By using Espresso, you can manually perform UI interactions in the app, and then create code that will replay those interactions automatically.

MAuto [46] is a tool designed to automate the testing of mobile games, which helps testers to create tests that are compatible with Android games. It utilizes image recognition and the Appium framework to record and play back user actions on any Android device. MAuto aims to reduce the amount of manual labor needed for testing and enhance the pace of test automation script generation. MAuto utilises both record and replay, as well as icon based GUI testing techniques.

Record-and-replay testing techniques in Android have limitations. For instance, these techniques can only test the scenarios that were previously recorded. This implies that if there is a new feature or functionality added to the app, the test script must be modified manually to include the new scenario. Moreover, record-and-replay testing may not be ideal for testing complicated or dynamic apps that have changing UI elements or rely on external data

sources.

LIT[47] is a tool that simplifies the process of testing games by automating the playtest.It generates a collection of abstract playtest tactics that are context-specific, indicating what actions can be performed under different circumstances by tactic generalization and tactic concretization.

3.2 Machine Learning based Android Game Testing

With the advancement in Machine Learning, some researchers have dived into the direction of making use of Machine Learning techniques to use for testing Android games.

DroidGamer[40] is a novel testing method for Android game apps that combines GUI traversal and deep learning models to identify functional GUI widgets. It introduces a new algorithm for navigating the GUI structure and employs deep learning models for widget recognition to determine the equivalence of GUI states. The approach involves training a faster R-CNN (Region-based Convolutional Neural Networks) model using annotated screenshots of a collection of game apps to recognize operable widgets. During the execution of the program, DroidGamer captures a game screenshot after triggering an event and feeds it to the trained deep learning model for operable widget detection. The model outputs a set of bounding rectangles representing the predicted positions of each operable widget. DroidGamer utilizes machine learning techniques specifically for identifying functional GUI widgets and does not provide information about the specific operations to be performed.

Wuji[50] is a testing system for online combat games that is automated and utilizes a combination of evolutionary algorithms, multi-objective optimization, and deep reinforcement learning. It trains a group of agents (i.e., deep neural networks) that can play the game

automatically, and these agents continuously update their policies to explore different states and complete missions. Wuji employs proposed oracles for on-the-fly testing while simultaneously training the agent policies.

ATGW[44] presents an innovative approach to creating an automation testing platform for a mobile game called Woody, leveraging machine learning algorithms. The system consists of three main components: a testing platform, a game state recognition algorithm, and game agents. The testing platform is built using the Airtest IDE, enabling testing across multiple devices. The game state recognition algorithm utilizes the Adaptive Gaussian Thresholding algorithm to identify the game board, and the Mean Square Error function is employed to predict the status of three blocks in each turn. The game agents are trained using reinforcement learning to mimic the playing behaviors of various users at three different skill levels. The experimental results demonstrate the effectiveness of both the game state recognition algorithms and game agents. This approach contributes to the research community by showcasing the application of machine learning in the mobile gaming industry.

Developers at King - the developer of Candy Crush Saga have started using Deep Reinforcement Learning techniques for testing their suite of candy crush games. They make use of it especially when new levels need to be added. For their approach, they use image encoding for input to the deep neural network. In image encoding, they represent the input as a 9x9 grid with 102 binary channels, each representing a game element and describing if it is present or not on each cell. This allows for faster and more efficient learning at the game-element level.After this, action encoding is applied wherein the one-hot encoded index of the edge between two cells is used [5]. This is then used to train the neural network.

In the works reviewed where deep reinforcement learning techniques are used, the major limitation is that these algorithms are developed with one game (like [44] and [5]) or a set of games ([50]) with very similar actions are used. This makes the reinforcment learning

agent getting trained in a similar style of actioning. It may get difficult to generalize these techniques to a broader range of games, hence making the adaptability less feasible. Another issue with these approaches is that anyone who wants to make use of similar techniques will need to have deeper understanding of ML concepts before being able to adapt these for their use case. In this paper, we create DRAG to address these limitations and develop a framework which can learn playing techniques for a wider range and category of games, and also make it feasible for use by developers who may have limited knowledge of ML frameworks.

Chapter 4

Motivating Example

There are over 500,000 Android games on the Google Play store today [27]. These games can belong to different genres like arcade, casual, racing, RPG, puzzles, etc. As of April 2023, Candy Crush is the second most played Android game [25]. . Candy Crush is a closed-source game, hence we looked at other games of similar design which were open-sourced. CasseBonbons [7] is an open-sourced game which has similar match-3 game playing logic as Candy Crush.

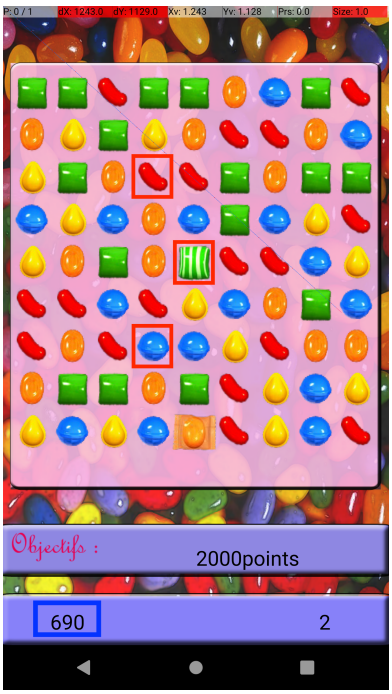

Figure 4.1: A snapshot of Cassebonbons

In this section, we look at Cassebonbons to explain our tool DRAG . This is a match-3 kind of game where the user needs to match 3 or more candies of the same type in a row or column to progress in the game. This game is a good testament of random testing tools like Monkey not doing as well, because these tools do not follow any game rules. DRAG analyses the actions taken by the user during the demo and uses these actions to play the game.

Demo Recording: For the game, developer provides certain inputs for the tool to use:

- Score Box: As shown in 4.1, the score box highlighted in blue is to be input by the user. Our tool provides a screenshot of the image and the developer can select the score box. This will give the coordinates of the score box which can be input

- Actionable Icons: These are all the icons which can be actioned on by the user. These are highlighted in 4.1 in red. These can be taken by the developer from a game screenshot, or from the assets folder because for many games, the developers need to specify these icons in that folder.

- Menu Icons: These are all the icons which have some meaning in the game like to restart the game, play new game, return to menu, etc.

Once these static inputs have been provided, the developer can play the game for a specific time period.

Action Analysis: In cassebonbon, the candy is swiped in one of four directions : left, right, bottom or top and swapped with another candy to make a match of 3 or more candies. The analysis of the video demo recording and action trace is perfomed by DRAG . During this, the starting coordinates x_1, y_1 are used to recognise if the action is being performed on one of the icons specified by the developer. Then for a swipe action, the end coordinates x_2, y_2

are used to calculate the swipe distance and swipe angle as follows:

$$swipe\ distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$\theta(radians) = \tan^{-1} \frac{y_2 - y_1}{x_2 - x_1}$$

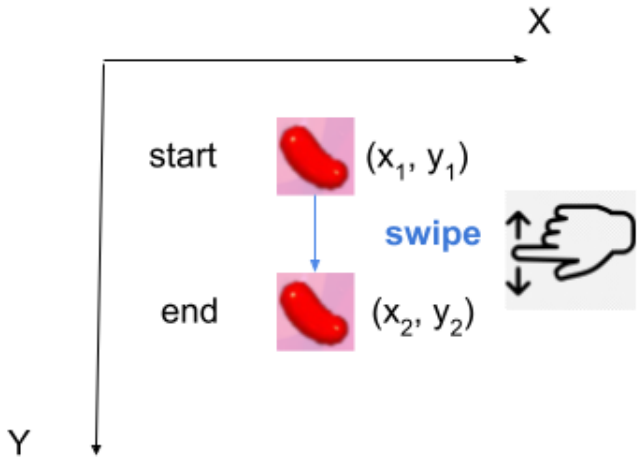

Figure 4.2: Swipe Action in Cassebonbons

For the entire duration of the demo, the actions are analysed for distance and angle to generate an action space to provide as an input to the reinforcement learning agent. The action analysis tool creates actions and enriches the information with icon usage.

Reinforcement Learning : Once the action analysis is performed, the action analysis generated previously is then translated to a form which is understood the RL agent - a list of all potential actions taken during the game. Each action is encoded to a value between 0 to 13, where each value has a meaning.

The Deep-Q network uses the action which has information about icon being used or not.

```
{
    "action_icon_used": true,
    "action_second": 15.488274000090314,
    "duration": 0.6356820000219159,
    "endX": 397,
    "endY": 1285,
    "icon_used": true,
    "id": 28,
    "startX": 425,
    "startY": 1104,
    "timestamp": 1677485574.8634584,
    "type": "swipe"
},
```


Figure 4.3: Action analysis of Cassebonbon

In case of Cassebonbon, an actionable icon is used. RL agent chooses an actionable icon from the list of all icons specified. This is recognised in the game scene using OpenCV which provides the centroid coordinates x_1, y_1 of the recognised icon. The swipe action is performed and the coordinates of the end point are computed as follows:

$$x_2 = x_1 + (distance * \cos\theta)$$

$$y_2 = y_1 + (distance * \sin\theta)$$

Chapter 5

Approach

DRAG is a tool created to automate testing of Android games. It consists of three distinct phases. 5.1 describes the high level architecture of the system. Sections 5.4.1 to **??** describe these phases in greater detail. Finally, section 5.5 details the integration of JaCoCo with the game code, which is vital to the tool as it is provided as a reward metric to the Reinforcement Learning.

5.1 System Overview

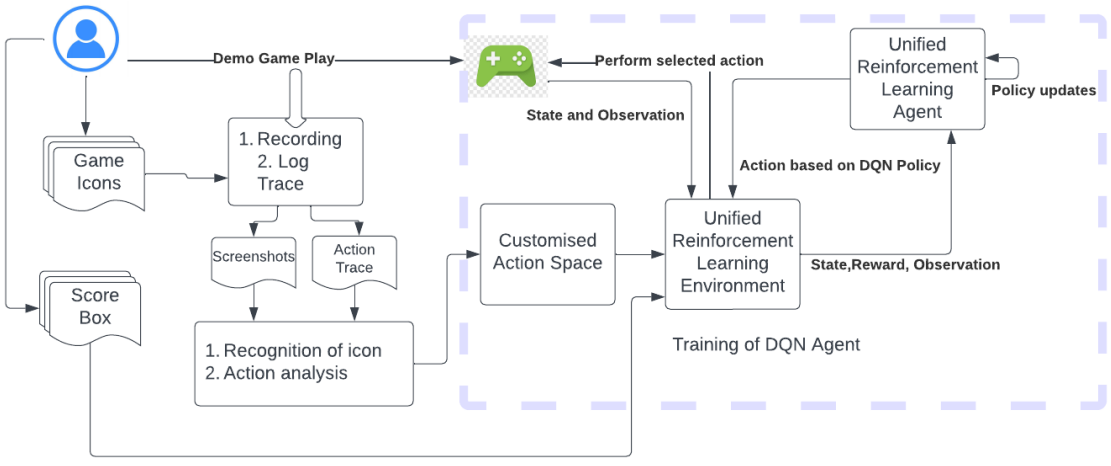

Figure 5.1: DRAG System Overview

DRAG consists of three major components:

- **Unified Reinforcement Learning Environment** consists of a generalised environment which interacts with game apk file, takes action on it, captures the states and observations and sends back reward to the agent.

- **Unified Reinforcement Learning Agent** consists of the deep neural network implementation of the Q-agent. It consists of all the layers inside the neural network, hyperparamaters regarding the learning rate, decay value, batch size, etc.

- **Customised Action Space** is the only component which is tailored for each game. To enable this, various manual inputs are provided by the developer including a demo video recording, analysis of the actions taken and inference of the action space for that particular game.

The next few sections explain the details of each component.

5.2 Unified Reinforcement Learning Agent

DRAG uses the PyTorch framework provided implementation of the DQN network. It makes use of 3 convolution 2D layers used with ReLU. It is then optimised using the RMS Prop optimizer. For balancing between the exploration and exploitation as the training progress, we use the value of epsilon to start at 0.9 and to end at 0.05. The decay factor used is 200. We use the Smooth L1 loss to compute loss of the model during training.

To compute the ideal time for training, we selected 6 games - one from each category defined in 6.1 and ran the agent for 1 hour. We observed the pattern of the coverage during the entire time, and capture the time after which the coverage converged and didn't change. The results of this run are shown in Table 5.1. It can be observed that each game stabilised it's

Game	Stable Time for max coverage (in minutes)
Frozenbubble	24
Open Flood	3
Cassebonbon	14
Snake	11
2048	8
Memory Game	21

Table 5.1: Time taken for coverage to stabilise for one game per category

coverage in under 25 minutes. To allow for some extra time in case some other game may need, we chose 30 minutes as the train time for our agent.

5.3 Unified Reinforcement Learning Environment

The unified learning environment interacts with the reinforcement agent.

5.3.1 Environment

The underlying environment used in DRAG is based on Gym. Any environment based on gym needs to implement the gym interface. This constitutes implementing the following methods:

- Constructor: The constructor helps setup anything which maybe passed to the class and may be required by the environment. For our implementation, we use the constructor to define the package name for the apk, classes path, apk path, score box, etc.

- Step: This is the method where the logic for taking an action resides. An action is passed to the step method. The agent performs the given action, calculates the reward

for the action performed and returns the observation, reward and also if this action terminates the agent episode. This is custom defined : in DRAG we use the coverage calculated and if it exceeds 90%, we conclude the episode.

- Reset: Reset method is invoked when a new episode begins to reset the environment. In DRAG , we use the reset method to uninstall the apk and install it again. This ensures the game begins at the same stage at the end of every episode for the training to be effective.

For the action space, we use discreet action space for each game. The action space is defined as a list of actions where each action is of the form:

$$Tap\ Action\ Tuple = (index, action\_id)$$

$$Swipe\ Action\ Tuple = (index, action\_id, \theta, swipe\ distance)$$

Here index is the index of this tuple in the list and action id denotes the specific action which is defined by DRAG . For instance, action_id 5 denotes a tap action whereas action_id 9 denotes a tap on an icon. These action ids are generated by the Action Inference tool and understood by the RL agent.

For swipe action, the start coordinates are determined by using OpenCV template matching for the icons specified for the game. The end coordinates are determined as follows:

$$x_2 = x_1 + (distance * \cos\theta)$$

$$y_2 = y_1 + (distance * \sin\theta)$$

5.3.2 Rewards

For any Reinforcement Learning model, it needs to be provided with a reward after each action is performed. In DRAG , we use a combination of two things as reward:

- Score Change: This is the difference between the scores before and after an action is performed on the game. For a game where score is not present, we assign the value of 0 to score change.

- Coverage Change: This is the difference between the coverage values before and after a step action is taken on the game. This coverage is generated using JaCoCo and will be explained in 5.5.

$$Reward = \delta_{coverage} + \delta_{score}$$

5.4 Customizable Action Space

To make DRAG effective, the Reinforcement Learning agent needs an action space that is not sparse and is smaller. To make action space tailored for each game, developers needs to provide certain manual inputs. On providing these inputs, the analysis of the actions is performed to generate an action space.

5.4.1 Manual Inputs

There are three different manual inputs required by DRAG from the developer. We will discuss each of these in detail.

5.4.2 Game Icons

For playing the games effectively, humans make use of their knowledge to interpret various icons present on the game screen. For a standard application like a browser, these icons may be generic like settings, back, forward, etc. But for games, these icons are custom-made and will hold specific game playing logic. This makes it essential to identify these game icons. For our case, we divide the game icons into two categories:

- Actionable Icons: These are set of all icons that can be actioned upon during the game play. Figure 5.2 shows all icons which can be acted on while playing the game Blockinger[4].
- Menu Icons: These are set of all icons which help in moving inside the game, i.e, to navigate between the different screens, to restart the game, to select a level, and so on. These are not as game specific as action icons in terms of their functionality, but the look and feel of these icons is also very game-specific and hence is required to be input. Figure 5.3 shows different menu icns which help in choosing a diffuclty level in the Memory Game [18].

5.4.3 Score Box

Majority of the games have a score associated with the game play to indicate the progression of the user and to move forward in the game. This game score can be a good indicator of not only the progress, but also the game code exploration. It directly comes from the idea that as a person achieves score in the game, new levels can unlock. This is the idea behind using score as a reward metric.

There were two main challenges due to which we resorted to introducing a manual input

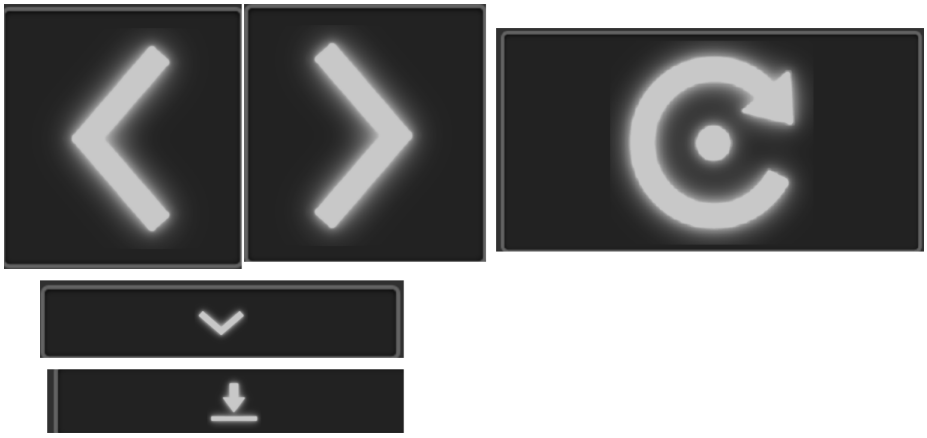

Figure 5.2: Actionable Icons in Blockinger

for the score box. First, the score can be in various parts of the screen in different games. Second, if we provide the entire screen for digit recognition, other numbers present on the screen can interfere with score and cause issues. To obtain score of the game manually, we created a Python script which renders a screenshot of the game. The developer can draw a box around the score and the script will output the coordinates of the bounding box. Figure 5.4 shows the sample output after selecting a bounding box around score in the game.

5.4.4 Demo Video

The reinforcement learning agent needs to have a defined action space to take actions during the game play. Initially, we decided to have a generic action space which included two actions for all games - tap and swipe. If the game screenshot is x pixels wide and y pixels high, there are x*y pixels to choose from as the starting point. This starting point can either be tapped on, or a swipe action can be initiated from. Again, the end point would have x*y - 1 possibilities. This makes the action space extremely large, and sparse because only very few actions would actually fetch reward (or even cause a change in the state).

To overcome this issue of a large and sparse action space, we decided to record a demo video

Figure 5.3: Menu Icons in Memory Game

```
Top left x:  95.0
Top left y:  53.0
Bottom right x:  187.0
Bottom right y:  144.0
```

Figure 5.4: Output after selecting the score box

by the developer and capture the action log. Both the action log and video were recorded using the adb shell. The demo time was selected as 8 minutes. From table 6.1, we chose one game from each category and manually played the game for 10 minutes capturing the coverage each minute. The analysis showed that coverage converged to its maximum value at 8 minutes or less for all the 6 games chosen. Hence, we configured the demo time to be 8 minutes. Table 5.2 shows the time taken during manual play for each game to reach a converged value of coverage.

A sample trace can be seen in Figure 5.5. "ABS_MT_TRACKING_ID" marks the start of an action. "ABS_MT_POSITION_X" and "ABS_MT_POSITION_Y" show the coordi-

Game	Time taken for coverage to converge
Open Flood	6m
FrozenBubble	8m
Cassebonbon	8m
Snake	2m
2048	6m
Memory Game	5m

Table 5.2: Time taken for coverage to converge during manual play

nates x and y for the action.The value in the first column (highlighted in red) indicates the timestamp.

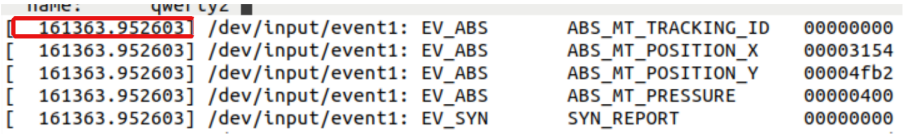

Figure 5.5: Snippet of the adb trace file

5.4.5 Action Inference

Once the manual inputs have been provided in Phase 1, we move to the next phase - where we use the manually played game demo and the adb generated log trace to infer the action space for the game.

- As shows in Figure 5.5, there is a column in the trace which denotes the timestamp of a particular action. We use this timestamp to go the particular point in the demo where the action was taken and grab a screenshot from the video. So, we convert from the video to time based screenshots and provide these screenshots as the input to our next step.

- In the next step, we use the coordinates (x,y) being actioned on. These coordinates

are also present in the trace as explained in section 5.4.4. We go thorugh all the menu icons provided, and discard this action if a match of (x,y) is found with a menu icon. This is done because these menu actions are not specific to a game, and generally just required to be tapped on. A similar menu icon matching is performed in Reinforcement Learning to handle this case. This is explained in detail in next section.

- If the (x,y) coordinates are not indicative of menu icon, we make use of the actionable icon list provided by the developer in Phase 1 (5.4.2). Template Matching provided by OpenCV can provide presence of all occurrences of an image (referred to as small image) in the bigger image. Once this list of all occurences is obtained for one icon, we use coordinates (x,y) to see if a match with the icon is found. The process is repeated for all icons unless a match is found. If no match with any actionable icon is found, we consider this as a case of a game where the icon usage is not relevant to the action being performed.

5.4.6 Action Space Generation

In addition to the information about icons, we need to also interpret the log to generate the actual action type - swipe, tap or combination.

1. Tap Action: For tap action, the only information needed are the coordinates of the tap. These are available from the log itself. 5.7 shows the snippet for the action analysis file generated by the Action Inference tool for a tap action. This shows the startX and startY coordinates for the action and also the duration of the tap.

2. Swipe Action: For swipe action, we need the start and end coordinates. 5.8 shows the snippet for the action analysis file generated by the Action Inference tool for a swipe

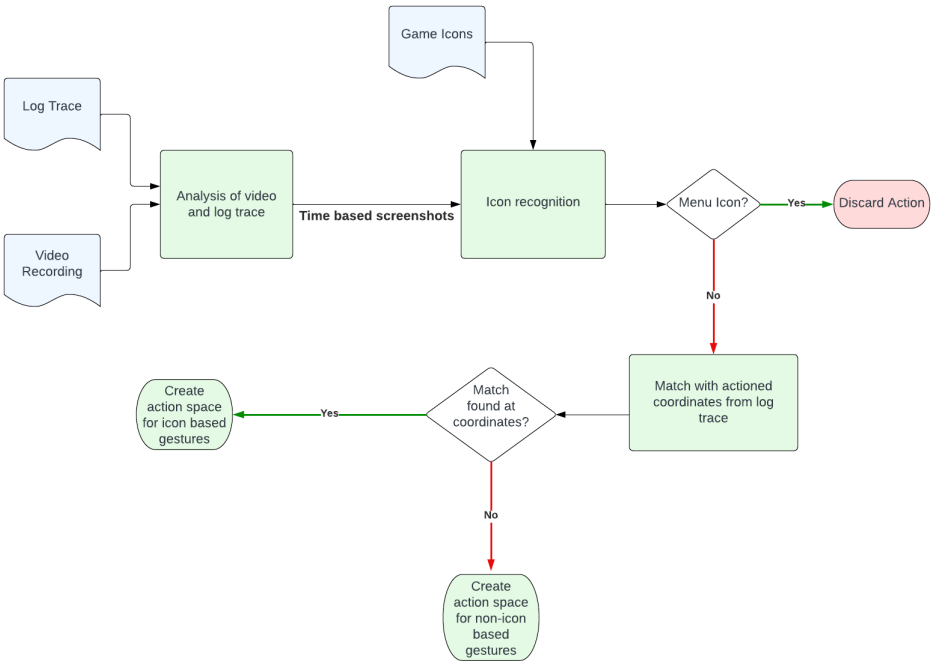

Figure 5.6: Design of Action Inference Tool

action. This shows the startX, startY, endX, endY coordinates for the action and also the duration of the swipe. We can gather two key information for each swipe - the distance of swipe and the angle of swipe.

$$swipe\ distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$\theta(radians) = \tan^{-1} \frac{y_2 - y_1}{x_2 - x_1}$$

Once this is done for each swipe action and n total swipe actions, we calculate the average swipe distance as:

$$distance_{avg} = \sum_{i=1}^{n} distance_i / n$$

For the angle theta, we find the maximum and minimum angles in the swipe action list, and also the average theta difference in the sorted angle list. We then go from the minimum angle to the maximum angle in steps if theta difference to generate the action space for a swipe angle.

$$sorted(\theta) = [\theta_0, \theta_1, .., \theta_n]$$

$$diff\ list = [\theta_1 - \theta_0, \theta_2 - \theta_1, ..., \theta_n - \theta_{n-1}]$$

$$k = length(diff\ list)$$

$$\theta_{avg} = \sum_{i=1}^{k} \theta_{diff[i]}/k$$

3. Combination Action: For some games, the action can be a combination of a couple of simple underlying actions. In our tool, we consider complex actions which consist of two underlying actions. In doing so, when we read the log, we look at the time difference between the actions. We make the assumption that if two actions are taken less than 1 second apart and are different from each other, they form a complex action. Our tool will combine the two actions to create an action space of two actions performed in succession to form one action.


```
{
    "action_second": 0.049740999995265156,
    "duration": 0.043684999996912666,
    "icon_used": false,
    "id": 1,
    "startX": 633,
    "startY": 708,
    "timestamp": 1675705202.4405143,
    "type": "tap"
},
```


Figure 5.7: Snippet of action file for a tap action

```
{
    "action_icon_used": true,
    "action_second": 15.488274000090314,
    "duration": 0.6356820000219159,
    "endX": 397,
    "endY": 1285,
    "icon_used": true,
    "id": 28,
    "startX": 425,
    "startY": 1104,
    "timestamp": 1677485574.8634584,
    "type": "swipe"
},
```


Figure 5.8: Snippet of action file for a swipe action

5.4.7 Special Considerations

Our tool relies on a demo for generating this action space. A manual input can also have erroneous actions which can add noise. For instance, a game which is primarily played by swiping can have a tap action in its trace which may have been performed mistakenly. To overcome this noise, we analyse all the actions. If the actions are not a complex action, and one kind of action dominates the action space, we discard the other action as being erroneous and noisy.

Another special case handling performed by the tool is to consider swiping in 4 directions as a separate case than usual swipe. If 70% of the actions have a theta as mentioned below (a difference of 5 degrees in each direction is considered as manual error), we consider these to be one of right, left, up or down swipes:

For right swipes,

$$-0.08 <= \theta <= 0.08$$

For up swipes,

$$1.48 <= \theta <= 1.65$$

For left swipes,

$$3.05 <= \theta <= 3.22$$

For down swipes,

$$-1.65 <= \theta <= 1.48$$

5.5 JaCoCo Integration

JaCoCo is a library used to capture coverage in terms of line coverage, branch coverage, class coverage, etc. We make use of JaCoCo to integrate with the game source code and then utilise it during game play to capture line coverage. Integrating JaCoCo requires some classes to be added to the source code and making changes to the AndroidManifest.xml file. This enables a coverage event to be emitted from a calling script which in turn generates a coverage.ec file. This file can then be used to open with Android Studio to compute coverage.

Chapter 6

Evaluation

In this chapter, we first define the dataset used for the experiment. Then we define the metrics we use for the evaluation of our tools. In the following sections, we discuss the different experiments performed to answer the research questions defined in Chapter 1.

6.1 Dataset

To evaluate various aspects of our tool, we chose 19 different open-sourced games. The reason for choosing open sourced games is that we needed line coverage for our tool, for both providing a reward as well as evaluating the line coverage achieved. This can be achieved only by integrating JaCoCo, which requires access to source code. Table 6.1 provides a list of all games selected, and also provides links to their respective repositories. The table also details category of each game. these categories have been assigned by us ans are based on the icon and gestures used in the games. For instance, Apple Flinger involves swiping the flinger towards the objects, which in turn makes the category as Icon based swipe.

6.2 Metrics

For evaluating DRAG , we use the following metrics:

S.No	Game	Category
1	Apple Flinger[2]	Icon based swipe
2	Blockinger[4]	Icon based tap
3	2048[11]	Swipe
4	Dodge[8]	Single Tap
5	Open Flood[22]	Single Tap
6	Frozen Bubble[10]	Icon based multiple gestures
7	Memory Game[18]	Icon based multiple gestures
8	Minesweeper[19]	Single Tap
9	Open Sudoku[23]	Multiple gestures
10	Pong[24]	Icon based swipe
11	Snake[30]	Icon based tap
12	Vector Pinball[31]	Icon based swipe
13	Cassebonbon[7]	Icon based swipe
14	Archery[3]	Icon based swipe
15	Linkup[15]	Icon based multiple gestures
16	Simple Brick Game[29]	Icon based tap
17	Privacy Friendly 2048[26]	Swipe
18	MemoGame[17]	Icon based multiple gestures
19	CandyMemory[6]	Icon based multiple gestures

Table 6.1: List of all games selected along with the respective action based categories

- Line Coverage defined by:

$$Line\ Coverage = \frac{\#\ lines\ covered}{\#\ total\ lines\ of\ code} * 100\%$$

- Score: Score achieved during the gameplay. For some games, this value may not be available.

6.3 Effectiveness of DRAG compared to other tools

To gauge the effectiveness of DRAG , we perform experiments to compare DRAG with two other tools - Monkey[20] and TimeMachine [34]. Monkey just generates random clicks which may or may not help in entering and using the application. TimeMachine is based

on generating different app states by taking actions on the UI elements and calculating coverage. It then uses states which increase coverage to generate further states and explore the application further.

For this experiment, we run all the three tools for 30 minutes. As explained in **??**, DRAG achieves converging line coverage in under 25 minutes when experimented with 1 game for each category (Table 5.1). To perform a fair comparison with tools, we run all the tools for 30 minutes and capture their performance in terms of score and line coverage.

Table 6.2 shows the compartive results on comparing DRAG with Monkey and TimeMachine. Any score value 0* indicates that the tool never entered the game or exited the game without playing, and thus, no score is available. For columns which contain - for score indicate that games do not provide score. For MemoGame, TimeMachine did not run at all, neither generated any error. We tried to run this multiple times, but it didn't work.

For three (Classic 2048, Minesweeper and Pong) of out of 19 games, Monkey has slightly better line coverage than DRAG . Even for these games, DRAG achieves a better score (wherever available). But, for the other 16, DRAG beats Monkey in both line coverage and score achieved. In all 19 games, DRAG does a better job than TimeMachine in terms of both the metrics.

On investigation of the the three games where Monkey performed better than DRAG and looking at the categories defined in table 6.1, two of the games (Classic 2048 and Minesweeper) belong to categories where identification of icons is not required. Any random action among swipe or tap can help play the game. Classic 2048 and Pong both do not have any menu icons, and starting the application starts the game without selecting any levels.

S.No.	Game	Monkey		Time Machine		DRAG	
		Score	Coverage	Score	Coverage	Score	Coverage
1	Apple Flinger	0	2	0	2	**1300**	**12**
2	Blockinger	0	10	0	1	0	**61**
3	Classic 2048	1412	**86**	1374	39	**1880**	78
4	Dodge	0*	42	0*	4		**55**
5	Open Flood	3	33	0	15	**19**	**46**
6	Frozen Bubble	-	46	-	35	-	**56**
7	Memory Game	1	53	0	2	**3**	**59**
8	Minesweeper	-	**49**	-	3	-	45
9	Open Sudoku	-	38	-	2	-	**39**
10	Pong	0	**67**	0	39	**26**	65
11	Snake	-	39	-	31	-	**44**
12	Vector Pinball	4000	51	500	29	**9400**	**52**
13	Cassebonbon	0*	4	0	4	**570(Level 2)**	**70**
14	Archery	0	31	0	10	0	**42**
15	Linkup	-	6	-	40	-	**56**
16	Simple Brick Game	0	18	100	15	**2200**	**41**
17	2048-variant two	0*	6	4	1	**76**	**51**
18	MemoGame	2	47	NA	NA	**4**	**55**
19	CandyMemory	-	11	-	8	-	**13**

Table 6.2: Comparison of DRAG with Monkey and TimeMachine for 30-minutes time

Finding 1 - On comparing DRAG with Monkey, DRAG achieves better line coverage for 16 of the total 19 games and achieves better score in all 19 games. When compared to TimeMachine, DRAG achieves better results in both line coverage and game score.

6.4 Effectiveness of DRAG compared to other tools when standard GUI widgets are also tested

Both Monkey and TimeMachine are based on taking random actions while testing an Android application. As seen from Finding 1 (section 6.3), Monkey performed better in achieving a higher line coverage for three games. To understand why this could potentially happen, we performed a variation of the experiment, where we introduced additional 10 minutes.

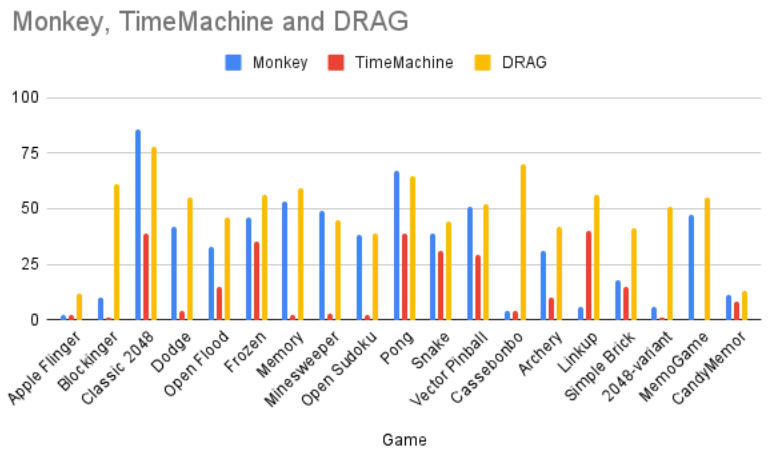

Figure 6.1: Comparison of DRAG , TimeMachine and Monkey

In these 10 minutes, DRAG would take random actions like Monkey and TimeMachine. To perform a fair comparison, we ran Monkey and TimeMachine for a total of 40 minutes each.

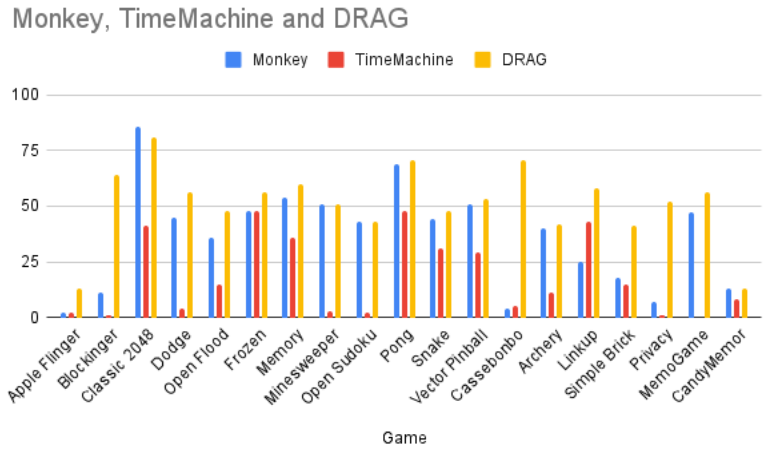

Figure 6.2: Comparison of DRAG , TimeMachine and Monkey when standard GUI widgets are also tested

Table 6.3 shows the results of running each of the three tools including DRAG for 40-minutes of playtime. There are a few observations which we can make from these results:

- For the three games (Classic 2048, Minesweeper and Pong) where Monkey performed

S.No.	Game	Monkey			Time Machine			DRAG		
		Score	Coverage	Diff	Score	Coverage	Diff	Score	Coverage	Diff
1	Apple Flinger	0	2	0	0	2	0	0*	**13**	1
2	Blockinger	0	11	1	0	1	0	0*	**64**	3
3	Classic 2048	0*	**86**	0	650	41	2	350	81	4
4	Dodge	0*	45	3	0	4	0		**56**	1
5	Open Flood	4	36	3	0	15	0	0*	**48**	2
6	Frozen Bubble	-	48	2	-	48	13	-	**56**	0
7	Memory Game	1	54	1	0	36	34	4	**60**	1
8	Minesweeper	-	**51**	2	NA	NA	NA	-	**51**	6
9	Open Sudoku	-	**43**	5	NA	NA	NA	-	**43**	4
10	Pong	0	69	2	0	48	9	26	**71**	6
11	Snake	-	44	5	NA	NA	NA	-	**48**	4
12	Vector Pinball	12600	51	0	NA	NA	NA	0*	**53**	1
13	Cassebonbon	0*	4	0	0	5	1	0*	**71**	1
14	Archery	0	40	11	0	11	1	0	**42**	0
15	Linkup	-	25	19	-	43	3	-	**58**	8
16	Simple Brick Game	0*	18	0	NA	NA	NA	0*	**41**	0
17	Privacy friendly 2048	0	7	1	NA	NA	NA	0*	**52**	1
18	MemoGame	1	47	0	NA	NA	NA	0*	**56**	1
19	CandyMemory	-	**13**	2	NA	NA	NA	-	**13**	0

Table 6.3: Comparison of DRAG with Monkey and TimeMachine for 40-minute time when 10-minute random actions are added to DRAG

better than DRAG , adding random actions to the tool playtime increased the coverage. In case of Pong and Minesweeper, the increase exceeds the coverage for the 30-minute gametime. In addition, for Pong, we can see that DRAG achieves better coverage than Monkey after 40-minutes, even though Monkey had a better coverage at the 30 minute mark.

- Due to the random nature of actions performed by Monkey and TimeMachine, the change in coverage is ranged from 0-19% in Monkey and 0-34% in TimeMachine. On the contrary, the variation in coverage for DRAG is only 0-8%, being mostly 0 or 1%. From this result we can infer that DRAG is able to achieve most of the coverage from taking game-specific actions and only some portion of it can get added by the additional random action play.

- For TimeMachine, 8 out of the 19 games couldn't run for the entirety of the 40-minute

run and we couldn't capture the results for the 40-minute playtime. These are denoted by "NA" in the Table 6.4.

Finding 2 - On adding an additional 10-minute gameplay time with random actions, DRAG is able to achieve as much coverage as Monkey in 2 of the 3 games. Overall, the increase in coverage for DRAG ranges from 0-8% whereas for Monkey the increase due to additional 10 minutes ranged from 0-19% and for TimeMachine ranged from 0-34%. Even with this increase, these tools could not exceed the coverage of DRAG in 18 of 19 games. This emphasises the fact that DRAG can achieve most coverage in a limited period with less increase in the additional random-action window.

6.5 Effectiveness of DRAG when no icon recognition is performed

To understand the effect of icon recognition in DRAG , we created two different variations of DRAG : 1) When only menu icons are recognised 2) When no icons are recognised. These two variations are important to study because from experiments performed in sections 6.3 and 6.4, we could see the importance of icon recognition because for many games, Monkey and TimeMachine were unable to even enter the game. For some games, where these tools could enter the game, they failed to play the game because the games required recognition of game icons.

Table 6.4 shows the comparison of the two variants of DRAG . For games like Snake, Vector Pinball, Pong the performance of the two variants is comparable. This is because there is no menu present for these games and launching them take us straight into the game. This is why both the variants perform equivalently. On the other hand, for games like Simple Brick

S.No.	Game	DRAG		DRAG - Menu Icons Only		DRAG - No Icons	
		Score	Coverage	Score	Coverage	Score	Coverage
1	Apple Flinger	**1300**	**12**	0	8	0	4
2	Blockinger	0	**61**	0	25	0*	15
3	Frozen Bubble	-	**56**	-	36	-	34
4	Memory Game	**3**	**59**	2	45	0*	23
5	Open Sudoku	-	**39**	-	36	-	7
6	Pong	**26**	**65**	0	56	0	56
7	Snake	-	**44**	-	40	-	37
8	Vector Pinball	**9400**	**52**	8200	51	0*	51
9	Cassebonbon	**570(Level 2)**	**70**	1240(Level 1)	53	0*	4
10	Archery	0	**42**	0	35	0	28
11	Linkup	-	**56**	-	41	-	30
12	Simple Brick Game	**2200**	**41**	0*	19	0*	15
13	MemoGame	**4**	**55**	2	47	1	40
14	CandyMemory	-	**13**	-	13	-	11

Table 6.4: Comparison of two variants of DRAG - one with only menu icon recognition and one with no icon recognition

Game, Casebonbons, there is a huge gap in line coverage. In these games, the menu icon recongition variant is atleast able to enter the game, but the variant with no icon recognition may or may not be able to enter the game.

For games like Memory Game, Cassebonbon, Pong even the menu-only variant don't perform poorly. This is becase the action space provided is still derived from the manual demo (as described in section 5.4.1). This means that even though the reinforcement learning agent doesn't have the information about which icon to perform a particular action on, it can still randomly act on the icon with the right action. This would still mean a better game progress than taking random action. Consequently, the line coverage doesn't drop drastically.

Finding 3 - Recognising icons is an imperative part of DRAG and is vital to it's performance. We cannot achieve at par results when we do not recognise icons in the game, and only perform game specific actions. Performing actions on specific icons provides the needed domain knowledge for the Reinforcement Learning agent to learn and progress in the game.

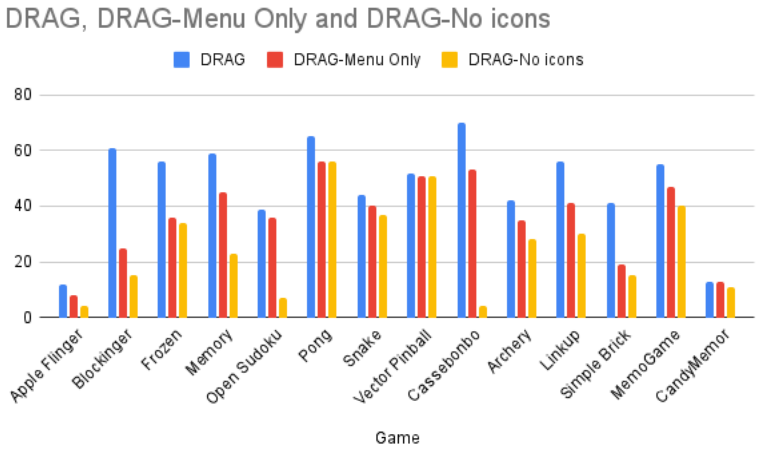

Figure 6.3: Comparison of different variations of DRAG

Chapter 7

Discussion

7.1 Extendability

DRAG tests 19 games which belong to different game genres and provide a wide variety of actions. In these 19 games, we identified 6 different action categories. A lot of other games which belong to similar action categories can be tested effectively with DRAG . Our scope was restriced because we could only perform experiments with open-sourced games.

Our work uses a dataset which is much larger than what has been previously explored in a lot of state-of-the-art works ([32], [50], [37]). In these games, the action set will be restricted or belong to only one type. In DRAG , we explored different action categories so that we could build a more generalizable approach. Any games which belong to action spaces in these 6 categories can then be tested.

Some categories of games may not be able to be tested effectively with DRAG . There are several reasons - 1) In the allocated demo time, we may be able to foresee only a subset of actions, which will then be supplied to the Reinforcement Learning algorithm. The new levels or game complexity may only uncover with a longer demo. 2) Some games include knowledge of general facts like quiz games, the alphabet (like Hangman) or even mathematics. The Reinforcement learning agent will not have any prior knowledge of this. This is also exemplified with the game OpenSudoku, where the game rule and knowledge are essential

to effectively play the game. 3) Games may also require Natural Language capacities, which will need to be integrated in order to understand the game playing logic.

7.2 Icon Recognition

Icon Recognition plays a vital role in the functioning of DRAG . This is demonstrated in section 6.5 where we created two different variation of DRAG and ran against the set of games where we were recognising game icons to begin with. In both the versions, the icon recognition played an important role, as it affected the performance of DRAG .

Providing game specific icons can be done easily because the game code contains the game icons in the resources folder. The only additional input required is regarding the game menu icons. These are also important because with no menu icon recognition, there are instances where the agent may not even be able to enter the game. This drastically affects the performance of the tool, and hence becomes an integral part.

In cases where the icons do not form a part of the game source code, these icons can be provided by taking screenshots from the game. Contrasted with the manual effort that may go in testing a game, this is not a significant amount of effort.

Chapter 8

Threats to Validity

During this research, we made several assumptions which can be threats to the validity of this work. We divide these threats into different categories: threats to external validity (section 8.1), internal validity (section 8.2) and construct validity (section 8.3).

8.1 Threats to External Validity

In this research, we experimented with 19 open sourced games. But there are over 500,000 games available on the Android Playstore. This can include both open-sourc ed and closed-sourced games. Also, the games we selected belong to 6 action categories. There are many more categories available. The results of this research may change when extended to other kinds of games and also close-sourced games. In addition, we integrate JaCoCo. We also omitted some games due to dependency issues with JaCoCo. These games may uncover some other insights which are not currently covered by this research.

8.2 Threats to Internal Validity

In the action inference tool, we made some assumptions to cater for human errors. These include removing erroneous actions based on if the action space consists of atleast 75% actions of one type and some other outlier actions of another type. Also, to generalise the

angles being taken during swipe actions, we converge actions between 5-degree difference of 90-degree angle in the four directions, we convert these to 90-degree. This is another assumption which may not be true in all games, and may require changes to the current approach.

In comparing our approach with TimeMachine, we run our experiments on emulator with API version 29, whereas TimeMachine can only run on emulator devices with API version 25. This may lead to some disparities in result because some games may not be compatible with lower Android version APIs.

8.3 Threats to Construct Validity

To perform a fair comparison of the tools, we captured the coverage files and screenshots of the games. The screenshots were captured at 1-minute intervals and the coverage files were captured at 30-minute and 40-minute marks. The screenshots were then manually analysed for capturing the score achieved. The coverage files were manually opened in Android Studio and the coverage captured. Both these manual processes can introduce errors despite the best efforts. This may also be a threat to the validity of the work.

Chapter 9

Future Work

9.1 Extend to other operating systems

DRAG was created to test Android Games effectively. But this solution can be extended to other operating systems in the future. Equivalent system to ADB needs to be found, and can then be included to work with other operating systems. This could potentially be a more extensive task. Also, for other operating systems, the main language of application source code may not be Java-based. JaCoCo works for Java-based code bases. If the code base is another programming language, we will need to find alternative ways to capture coverage.

9.2 Expansion of Game Dataset

The dataset currently includes 19 open-sourced games. This dataset can be expanded to include other open-sourced games which add more variety. This may mean the extension of the action inference tool as well.

Also, to expand to other games, the efficiency of DRAG may need to be improved. Currently, the tool takes a screenshot of the game after each action to analyse the score. It also captures the coverage before and after each step. This can add latency, which may become a bottleneck for fast-paced games. Future works may focus on this aspect.

9.3 Transfer Learning

Another important aspect which can be covered in future works relates to transfer learning. It will be interesting to see how a model trained on one game tests for another game which is similar. Many variations for the same game may exist. We can train the RL-agent for one game, and then apply the trained model to the other game. This can help in investigating if creating one agent can test multiple games which use the same logic and action space.

Chapter 10

Conclusion

Android games are gaining popularity and the numbers of Android games is increasing. In the end of Q3 2022, there were close to 500,000 Android games in the market. With the increasing numbers, it becomes essential to maintain the quality of the games. Better quality ensures that Game developers and organizations do not suffer from reputational and financial losses.

In this research, we create DRAG which tests Android games. The key distinguishing features of the tool include - generalizability in terms of extending to Android games outside of the dataset explored. Developers can provide minimal inputs and onboard the game to the tool. DRAG automatically detects the action space from an 8-minute demo video and corresponding action trace from the ADB tool. This action space include information about the actions like swipe or tap or combination and information on whether each action was performed on a specific game icon.

Once this action space is generated, we train a DQN agent to learn to play the game. In this process, we provide score and coverage change as a reward. Game screenshot is provided as a state and upon taking an action, the final observation is also provided as a game screenshot. The DQN agent keeps taking random actions initially, but with each step and corresponding reward, the greedy policy starts choosing more action from it's memory to fetch maximum rewards.

To asses the effectiveness of DRAG , we perform different experiments and compare the results with two existing tools - Monkey and TimeMachine. In the first experiment, we run all the three tools for a duration of 30 minutes, and capture the line coverage and score achieved. In this, DRAG performs better than TimeMachine in all games - achieving a better line coverage and score. For 16 of the 19 games, it performs better than Monkey.

To analyse the performance impact of taking random actions, we perform a second experiment where we add a 10-minute window to DRAG where it can take a random swipe or tap action. The results of this experiment demonstrate that DRAG is able to match with Monkey in terms of line coverage and score for 2 of the 3 games it could not perform as well in previous experiment.

Finally, we perform a study with two variations of DRAG - one where it only recognises menu icons and one where it does not recognise any icons. This study is performed to gauge the importance of icon recognition in the effectiveness of DRAG . This experiment shows that both variations perform worse than the original version where icons are recognised. This exhibits the vitality of icon recognition to the success of our tool.

With this study, we can conclude that Reinforcement Learning, specifically Deep Reinforcement Learning can be leveraged for the automation of testing of games. It can leverage domain knowledge from the icons supplied, the demo video provided and the score input for the game.

Bibliography

[1] Adb. https://developer.android.com/tools/adb.

[2] Apple flinger. https://gitlab.com/ar-/apple-flinger.

[3] Archery. https://github.com/kalina2002/Archery.

[4] Blockinger. https://github.com/tasioleiva/Blockinger.

[5] Human like playtesting with deep learning. https://medium.com/techking/human-like-playtesting-with-deep-learning-92adafffe921, .

[6] Candy memory. https://github.com/tube42/candymem, .

[7] Cassebonbon. https://github.com/IsmaelCussac/casseBonbons.

[8] Dodge. https://github.com/dozingcat/dodge-android.

[9] Espresso. https://developer.android.com/studio/test/other-testing-tools/espresso-test-recorder.

[10] Frozen bubble. https://github.com/robinst/frozen-bubble-android.git.

[11] Classic 2048. https://github.com/tpcstld/2048.git.

[12] Gymnasium. https://gymnasium.farama.org.

[13] Jacoco. https://www.jacoco.org/jacoco/trunk/doc/.

[14] Junit. http://www.junit.org/index.htm.

[15] Link up. https://github.com/csuyzb/AndroidLinkup.

[16] Android market share 2023. https://www.bankmycell.com/blog/android-vs-apple-market-share/.

[17] Memogame. https://github.com/SecUSo/privacy-friendly-memo-game, .

[18] Memory game. https://github.com/sromku/memory-game/, .

[19] Minesweeper. https://github.com/SecUSo/privacy-friendly-minesweeper.

[20] Android monkey. https://developer.android.com/studio/test/monkeyrunner.

[21] Opencv. https://opencv.org/about/, .

[22] Open flood. https://github.com/GunshipPenguin/open_flood, .

[23] Open sudoku. https://github.com/ogarcia/opensudoku, .

[24] Pong. https://github.com/catalinc/pong-game-android.

[25] Top 20 android games. https://www.appbrain.com/apps/popular/games/.

[26] Privacy friendly 2048. https://github.com/SecUSo/privacy-friendly-2048.

[27] Android games report. https://www.statista.com/statistics/780229/number-of-available-gaming-apps-in-the-google-play-store-quarter/.

[28] Selenium. https://www.selenium.dev/.

[29] Simple brick game. https://github.com/TobiasBielefeld/Simple-Brick-Games.

[30] Snake. https://android.googlesource.com/platform/development/+/master/samples/Snake/src/com/example/android/snake.

[31] Vector pinball. https://github.com/dozingcat/Vector-Pinball.

[32] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. Automated video game testing using synthetic and humanlike agents. *IEEE Transactions on Games*, 13(1):50–67, 2021. doi: 10.1109/TG.2019.2947597.

[33] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103, 2007. doi: 10.1109/FOSE.2007.25.

[34] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20, pages 1–12, 2020.

[35] Yuning Du, Chenxia Li, Ruoyu Guo, Xiaoting Yin, Weiwei Liu, Jun Zhou, Yifan Bai, Zilin Yu, Yehua Yang, Qingqing Dang, and Haoshuang Wang. PP-OCR: A practical ultra lightweight OCR system. *CoRR*, abs/2009.09941, 2020. URL https://arxiv.org/abs/2009.09941.

[36] Sarah E. Garcia. Usability testing: Creative techniques for answering your research questions. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI EA '20, page 1–2, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368193. doi: 10.1145/3334480.3375064. URL https://doi.org/10.1145/3334480.3375064.

[37] Stefan Freyr Gudmundsson, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartlomiej Kozakowski, Richard Meurling, and Lele Cao. Human-like playtesting with deep learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018. doi: 10.1109/CIG.2018.8490442.

[38] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*,

AST '11, page 77–83, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305921. doi: 10.1145/1982595.1982612. URL https://doi.org/10.1145/1982595.1982612.

[39] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, 7: 133653–133667, 2019. doi: 10.1109/ACCESS.2019.2941229.

[40] Bo Jiang, Wenlin Wei, Li Yi, and W.K. Chan. Droidgamer: Android game testing with operable widget recognition by deep learning. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 197–206, 2021. doi: 10.1109/QRS54544.2021.00031.

[41] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. Record and replay for android: Are we there yet in industrial cases? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 854–859, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3117769. URL https://doi.org/10.1145/3106237.3117769.

[42] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931054. URL https://doi.org/10.1145/2931037.2931054.

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

[44] Thuy Pham, Nhu Nguyen, Tien Dang, Linh Nguyen, and Binh Nguyen. *ATGW: A Machine Learning Framework for Automation Testing in Game Woody*. 09 2020. ISBN 9781643681146. doi: 10.3233/FAIA200586.

[45] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[46] J. Tuovenen, M. Oussalah, and P. Kostakos. Mauto: Automatic mobile game testing tool using image-matching based approach. *The Computer Games Journal*, 8(3):215–239, Dec 2019. ISSN 2052-773X. doi: 10.1007/s40869-019-00087-z. URL https://doi.org/10.1007/s40869-019-00087-z.

[47] Y. Zhao, E. Tang, H. Cai, X. Guo, X. Wang, and N. Meng. A lightweight approach of human-like playtest for android apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 309–320, Los Alamitos, CA, USA, mar 2022. IEEE Computer Society. doi: 10.1109/SANER53432.2022.00047. URL https://doi.ieeecomputersociety.org/10.1109/SANER53432.2022.00047.

[48] Yu Zhao, Brent E. Harrison, and Tingting Yu. Dinodroid: Testing android apps using deep q-networks. *ArXiv*, abs/2210.06307, 2022.

[49] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards black box testing of android apps. In *2015 10th International*

Conference on Availability, Reliability and Security, pages 501–510, 2015. doi: 10.1109/ARES.2015.70.

[50] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 772–784. IEEE Press, 2020. ISBN 9781728125084. doi: 10.1109/ASE.2019.00077. URL https://doi.org/10.1109/ASE.2019.00077.