AUTOMATED VIDEO GAME TESTING USING REINFORCEMENT LEARNING AGENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

SİNAN ARIYÜREK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2022

Approval of the thesis:

AUTOMATED VIDEO GAME TESTING USING REINFORCEMENT LEARNING AGENTS

Submitted by Sinan Arıyürek in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Information Systems Department, Middle East Technical University** by,

Date: **14.09.2022**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: **Sinan Arıyürek**

Signature :

ABSTRACT

AUTOMATED VIDEO GAME TESTING USING REINFORCEMENT LEARNING AGENTS

Arıyürek, Sinan

Ph.D., Department of Information Systems

Supervisor: Assoc. Prof. Dr. Elif Sürer

Co-Supervisor: Assoc. Prof. Dr. Aysu Betin Can

September 2022, 91 pages

In this thesis, several methodologies are introduced to automate and improve video game playtesting. These methods are based on Reinforcement Learning (RL) agents. First, synthetic and human-like tester agents are proposed to automate video game testing. The synthetic agent uses test goals generated from game scenarios, and the human-like agent uses test goals extracted from tester trajectories. Tester agents are derived from Sarsa and Monte Carlo Tree Search (MCTS) but focus on finding defects, while traditional game-playing agents focus on maximizing game scores. Second, various MCTS modifications are proposed to enhance the bug-finding capabilities of MCTS. Third, to improve playtesting *developing persona* is introduced, enabling development in an agent's personality and more robust playtesting. RL algorithms aim to find paths that maximize the total accumulated reward. However, discovering various alternative paths that achieve the same objective is equally essential for playtesting. Consequently, Alternative Path Finder (APF) is introduced to let RL agents discover these alternative paths. We experiment with our proposed methodologies using the General Video Game Artificial Intelligence (GVG-AI) and VizDoom frameworks. The experiments reveal that human-like and synthetic agents compete with human testers' bug-finding performances. Furthermore, the experiments indicate that MCTS modifications improve bug-finding performance. Moreover, the experiments show that developing personas provide better insight into the game and how different players would play. Lastly, the alternative paths found by APF are presented and reasoned why traditional RL agents cannot discover those paths.

Keywords: Automated Game Testing, Automated Playtesting, Reinforcement Learning, Monte Carlo Tree Search, Player Modeling, Graph Coverage

ÖZ

PEKİŞTİRMELİ ÖĞRENME YÖNTEMLERİ KULLANARAK OYUNLARIN OTOMATİK TEST EDİLMESİ

Arıyürek, Sinan

Doktora, Bilişim Sistemleri Bölümü

Tez Yöneticisi: Doç. Dr. Elif Sürer

Ortak Tez Yöneticisi: Doç. Dr. Aysu Betin Can

Eylül 2022, 91 sayfa

Bu tez çalışmasında bilgisayar oyunu test yöntemlerini otomatize etmek ve geliştirmek için birçok metodoloji tanıtılmaktadır. Bu yöntemler, Pekiştirmeli Öğrenme (RL) ajanlarına dayanmaktadır. Öncelikle, video oyunu testini otomatize etmek için sentetik ve insan-benzeri test ajanları önerilmiştir. Sentetik ajan, oyun senaryolarından oluşturulan test hedeflerini kullanmaktadır. İnsan-benzeri ajan ise test edenin izlediği yollardan çıkarılan test hedeflerini kullanmaktadır. Bu test ajanları, Sarsa ve Monte Carlo Ağaç Araması (MCTS) kullanılarak türetilmekledir. Geleneksel oyun oynama ajanları oyun skorlarını maksimize etmeye odaklanmışken, test ajanlarının odağı hataları bulmaya yöneliktir. İkinci olarak, MCTS'nin hata bulma yetisini geliştirmek için çeşitli MCTS modifikasyonları önerilmiştir. Üçüncü olarak, oyun testini iyileştirmek için gelişen karakter tasarımı sunulmuştur. Gelişen karakter, bir ajanın kişiliğini geliştirmeye olanak sağlar ve böylece daha doğru oyun testleri elde edilir. RL algoritmalarının amacı toplam ödülü maksimize edecek yolları bulmaktır. Bununla birlikte aynı hedefe ulaşan alternatif yolları keşfetmek de oyun testi için önemlidir. Bu nedenle, RL ajanlarının bu alternatif yolları da keşfedebilmesi için, Alternatif Yol Bulucu (APF) tanıtılmıştır. Önerdiğimiz metodolojileri Genel Video Oyun Yapay Zeka (GVG-AI) ve VizDoom ortamlarını kullanarak test etmekteyiz. Deneyler, insan-benzeri ve sentetik ajanların, gerçek insanların yaptıkları testlerdeki hata bulma performanslarıyla rekabet ettiğini ortaya koymaktadır. Ayrıca deneyler, hata bulma performansının MCTS modifikasyonları sayesinde iyileştirildiğini göstermektedir. Dahası, deneyler gelişen karakterin oyuna ve farklı oyuncuların oyunu nasıl oynayacağına dair daha iyi bir fikir sağladığını göstermektedir. Son olarak, APF tarafından bulunan alternatif yollar sunulmuş ve geleneksel RL ajanlarının neden bu yolları bulamadığı açıklanmıştır.

Anahtar Kelimeler: Otomatik Oyun Testi, Otomatik Oyun Testi, Pekiştirmeli Ögrenme, Monte Carlo Ağacı Arama, Oyuncu Modelleme, Grafik Kapsamı

To my wife, and my family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisors, Dr. Aysu Betin Can and Dr. Elif Surer, for their support, understanding, patience, and supervision throughout my studies. I always appreciated their determination to realize this study, which once was no more than a fragment of our thoughts and hopes. I am glad to see that our ideas came to fruition, were acknowledged by others, and influenced further studies.

I would like to thank Dr. Ebru Aydın Göl and Dr. Erhan Eren for their invaluable guidance during my Ph.D. thesis. Furthermore, I owe my genuine thanks to our testers, who painstakingly tested out our games.

I am deeply grateful to my wife and my family; Bengü Kevinç Arıyürek, Cemre Arıyürek, Sezin Arıyürek, and Macit Arıyürek; for their understanding and support throughout my studies. Furthermore, I cannot credit Nala enough, our cat, who accompanied me during my studies. I am thankful to Bilinç Kevinç, Nuray Kevinç, Kahraman Kevinç, Yalım İşleyici, Simin İşleyici, Meltem İşleyici, and Hasan İşleyici for their support during my graduate studies.

I would like to thank my friends and my colleagues Serkan, Deniz, Can, Tankut, Uğur, Seda, Seher, Elif, Begüm, Umut, Marco, Onur, Burak, Duygu, Cihan, Sezin, Bertan. This journey would not have been possible without them.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ vi

DEDICATION viii

ACKNOWLEDGMENTS ix

TABLE OF CONTENTS x

LIST OF TABLES xiv

LIST OF FIGURES xv

CHAPTERS

1 INTRODUCTION 1

1.1 Research Questions 6

1.2 Contributions of the Study 7

1.3 Organization of the Thesis 7

2 PRELIMINARIES 9

2.1 Reinforcement Learning 9

2.2 Monte Carlo Tree Search 10

2.3 GVG-AI 11

2.4 VizDoom 11

2.5 Graph Testing 11

3 LITERATURE REVIEW 13

3.1 Game testing 13

3.2 Game Playing 14

3.3 Learning From Humans 14

3.4 MCTS Modifications 15

3.5 Playtesting 16

3.6 Personas in Playtesting 16

3.7 Automated Playtesting 17

3.8 Exploration Methods in Reinforcement Learning 17

4 METHODOLOGY I: TEST STATE 19

5 METHODOLOGY II: TEST GOAL GENERATION 23

5.1 Synthetic Test Goals 23

5.2 Human-Like Test Goals 25

5.3 Generating Test Sequences 27

6 METHODOLOGY III: MCTS MODIFICATIONS 29

6.1 Transpositions 29

6.2 Knowledge-Based Evaluations 29

6.3 Tree Reuse 29

6.4 MixMax 30

6.5 Boltzmann Rollout 30

6.6 SP-MCTS 31

6.7 Computational Budget 31

7 METHODOLOGY IV: ALTERNATIVE PATH FINDER 33

7.1 Measuring Similarity 33

7.2 From Recoding Probability to Intrinsic Feedback 34

7.3 From Predicting Dynamics to Intrinsic Feedback 35

7.4 APF Architecture 36

8 METHODOLOGY V: DEVELOPING PERSONA 39

9 EXPERIMENTS 43

9.1 Experiments for Synthetic and Human-like Test Goals 43

9.2 Experiments for MCTS Modifications 50

9.3 Experiments for Developing Persona and APF 51

10 RESULTS 59

10.1 Results for Synthetic and Human-like Test Goals 59

10.1.1 Experiment 1: Agents Testing Game A 62

10.1.2 Experiment 2: Agents Testing Game B 62

10.1.3 Experiment 3: Agents Testing Game C 63

10.2 Results for MCTS Modifications 63

10.2.1 Game A 63

10.2.2 Game B 63

10.2.3 Game C 65

10.3 Results for Developing Persona and APF 65

10.3.1 Experiment I: Procedural vs Goal-based personas: 65

10.3.2 Experiment II: Alternative paths found in GVG-AI: 67

10.3.3 Experiment III: Personas in Doom: 69

10.3.4 Experiment IV: Alternative paths found in Doom: 70

11 DISCUSSION 73

12 CONCLUSION & FUTURE WORK 79

REFERENCES 83

APPENDICES

A CURRICULUM VITAE 91

LIST OF TABLES

Table 1 Utility weights for the goals. 41

Table 2 The MCTS Agents' Modifications. 52

Table 3 Hyperparameters of RL Agents. 56

Table 4 Hyperparameters of APF Methods. 56

Table 5 Utility weights of game events for the procedural personas. 57

Table 6 Utility weights of game events for developing persona goals. 57

Table 7 Development sequences for developing personas. 57

Table 8 Goal criteria for developing personas. 58

Table 9 Bug Finding Percentage and Trajectory Length of Human Testers, Sarsa(λ), and MCTS. 66

Table 10 Cross-Entropy Results of Sarsa(λ) and MCTS. 67

Table 11 Interactions performed by the PPO RL agent in Experiment I. 67

Table 12 Interactions performed by the PPO + CTS RL agent in Experiment I. 68

Table 13 Total Discounted Reward without APFCTS and with APFCTS. 68

Table 14 Interactions of Personas in Experiment III over 1000 evaluations. 70

Table 15 Total Discounted Reward without APFICM and with APFICM in Doom. 71

LIST OF FIGURES

Figure 1 A game scenario graph. 3

Figure 2 Realized nodes of Game Scenario Graph from Figure 1. 3

Figure 3 An example VDGL snippet with SpriteSet and InteractionSet. 21

Figure 4 Game State and Test State 22

Figure 5 CTS Filters. 35

Figure 6 Alternative Path Finding Architecture. 37

Figure 7 An example level created by GVG-AI framework. 40

Figure 8 Developing Persona. 41

Figure 9 Four levels of Game A. 44

Figure 10 Four levels of Game B. 45

Figure 11 Four levels of Game C. 46

Figure 12 VGDL describing the ruleset for Level 1 of Game A. 47

Figure 13 VGDL describing the ruleset for Level 1 of Game B. 48

Figure 14 VGDL describing the ruleset for Level 1 of Game C. 49

Figure 15 Map of the first testbed game. 53

Figure 16 Doom in-game snapshot. 54

Figure 17 Map of second testbed game. 55

Figure 18 Map of third testbed game. 55

Figure 19 The Percentage of Bugs Found by RL Agents and Human Testers. 60

Figure 20 Percentage of Bugs Found by Human-Like RL Agents and Human Testers. 60

Figure 21 Cross-Entropy of Human-Like RL Agents. 61

Figure 22 Length of Trajectory/Sequence The RL Agents and Human Testers. 61

Figure 23 Number of Splits Made by MGP-IRL to Collected Human Trajectories. 62

Figure 24 Actions that causes faults in Level 2 of Game B. 64

Figure 25 Paths found by Exit persona with PPO and with PPO + CTS + APFCTS. 69

Figure 26 Paths found by Exit persona with PPO and with PPO + ICM + APFICM in GVGAI. 71

CHAPTER 1

INTRODUCTION

The video game industry is a growing multi-billion industry [50] where developing a successful game is a challenging task. The challenge arises because video game development combines multiple crafts[53], gamers are difficult to please [16] and span several platforms. Though the success of a video game can be attributed to numerous aspects, the game developers make every effort to produce the perfect game as possible. Therefore, game development companies conduct alpha and beta playtesting sessions to understand how gamers perceive the game. These playtesting sessions are indispensable as playtesters provide essential information such as the strong and weak points of the game and the bugs they have found. While human playtesting is beneficial for games, it introduces additional costs and latency to the game development process. Additionally, the requirements of the game change frequently [80], which requires repeating the tests while conducting new tests.

We can playtest a video game to find *bugs*. In this thesis, we define bugs as the discrepancy between game design attributes and game implementation. Researchers proposed several methodologies to automate game testing such as record/replay [64], handcrafted scenarios [19], UML and state machines [40], Petri nets [34], and RL [71, 51]. However, when the game environment changes, test sequences and scenarios such as [64, 19] become obsolete, and a manual tester effort is required to create a new test sequence. Unlike these manual techniques, UML-based techniques automate the test generation process. However, generating sequences from UML runs into state explosion for larger games; without a gameplay AI, it relies on the generated states to play the game. Hence, researchers employed AI to test games. A Petri net that contains high-level actions of a game is used to generate test sequences using an AI [34]. RL is used with short and long-term memory to test Point-and-Click games [71]. Furthermore, RL is used to test the crafting system in Minecraft [51]. More generalized approaches to test video games are used to find bugs in two GVG-AI games [52]. As the authors [52] have stated, since the agent was a gameplaying agent, the agent was not required to find all the bugs. Lastly, a team of agents with different purposes is introduced to test games [32]. Nevertheless, these studies have at least one of the following drawbacks no controlled testing environment, no automated oracle, and no RL agent.

In our first paper [5], we showed how we could generate human-like and synthetic test goals and then use RL agents to realize these goals. Test goals are objectives that agents want to validate in a game. When we exercised these agents, they generated trajectories that are checked by an automated oracle to detect bugs. We used RL agents, as automated game testing frameworks fail to perform without an RL agent when the game under test is updated. Furthermore, in the last decade, researchers have shown that RL agents can surpass humans in arcade games [55], Go [85], StarCraft II [96] and Dota 2 [60]. However, these RL agents were game-playing agents, and the requirements for game testing agents

are different. For example, a game testing agent will also test the failure scenarios to verify whether failing is possible. Nonetheless, this scenario would be a training failure for a game-playing agent. Consequently, we proposed two ways to encapsulate these testing goals, human-like and synthetic, which an RL agent can realize as test scenarios.

Human-like agents tend to perform actions similar to humans. Therefore, these agents are used to create believable opponents or to playtest a game like a human. Researchers used human-like agents to create believable opponents and generate human-like playtest data in games such as Unreal Tournament [92], Super Mario [61], Catan [23], GVG-AI [45], Spades [22], and Candy Crush [31]. Inverse reinforcement learning (IRL) [59] is one of the ways to extract human-like objectives from collected human playtest data. In game playing, most human players stick to the same goal: winning the game. However, for game testing, testers may experiment with certain test goals and then test another test goal. Consequently, the IRL approaches such as Apprenticeship Learning [1], Maximum Entropy IRL [101], Apprenticeship Learning About Multiple-intentions [7], and Maximum Entropy Deep IRL [97] may not explain the structure behind observed tester behavior. There are approaches such as Bayesian Non-parametric IRL [54] and IRL via Spatio-temporal Subgoal Modeling [88]; these approaches fall short of generalizing to unseen states and the ability to generalize to different levels, respectively. Consequently, we proposed the Multiple-Greedy Policy IRL (MGP-IRL) [5] to explain the structure behind collected human test trajectories. In this thesis, we refer to the SARSA or MCTS agent that realizes extracted human test goals as the human-like agent.

On the other hand, human-like agents incur the human-playtest data. Consequently, the game must have been playtested before. We proposed synthetic test goals to offer an alternative to human-like test goals. Synthetic goals are handcrafted goals by an expert and researchers used these goals within RL agents to play arcade games [55], role-playing games [36], and Match-3 games [56]. For gameplaying handcrafting an objective is simpler than handcrafting an objective for testing as testing requires interaction with a more extensive set of game elements. We tackled the synthetic test goal generation from the perspective of software testing. A game can be viewed as implementing the game designer's story. This story —whether linear or non-linear— can be represented using a graph [2]. In this thesis, this graph is referred to as a game scenario graph. The game scenario graph (see Figure 1) is designed by the game designer and contains high-level behavior. The nodes on this figure hold atomic properties that match the game properties or sprites. The available transitions between the nodes are shown using arrows, and the nodes that do not have an outgoing arrow are terminal nodes. A node on this graph is matched with the states of the game (see Figure 2). Edges are the actions that progress the story. Additionally, as directed graphs form the foundation of several coverage criteria in software testing [3], it is possible to generate test sequences using this graph and a coverage criterion. We enhance the test sequence by adding actions at each node that should not progress the game, such as colliding with an obstacle. The coverage criterion determines how much of that objective must be tested. For example, if the objective is to collide with obstacles, coverage refers to how many of the obstacles the agent should collide with. Consequently, the test sequences verify the implementation of the game scenario while the enhancements check other aspects of the game such as testing collisions and unintended actions. In this thesis, we refer to the Sarsa or MCTS agent that realizes synthetic test goals as the synthetic agent.

In our paper titled "Automated Video Game Testing Using Synthetic and Human-like Agents," we used SARSA and MCTS agents [5] to realize human-like and synthetic test goals and compared these agents against human testers. We used the GVG-AI framework to craft three different games, each of four levels. GVG-AI games are written in Video Game Description Language (VGDL). Later this VGDL is

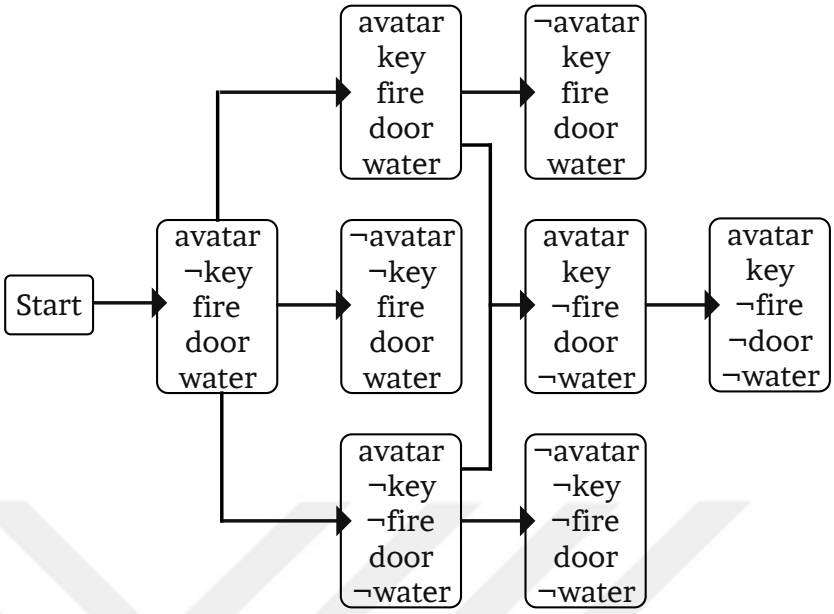


Figure 1: ©2019 IEEE.
A game scenario graph using the predicates $avatar$, key, $fire$, $door$, and $water$. key denotes whether avatar carries the key. The other predicates denote whether these sprites exist in the current frame.

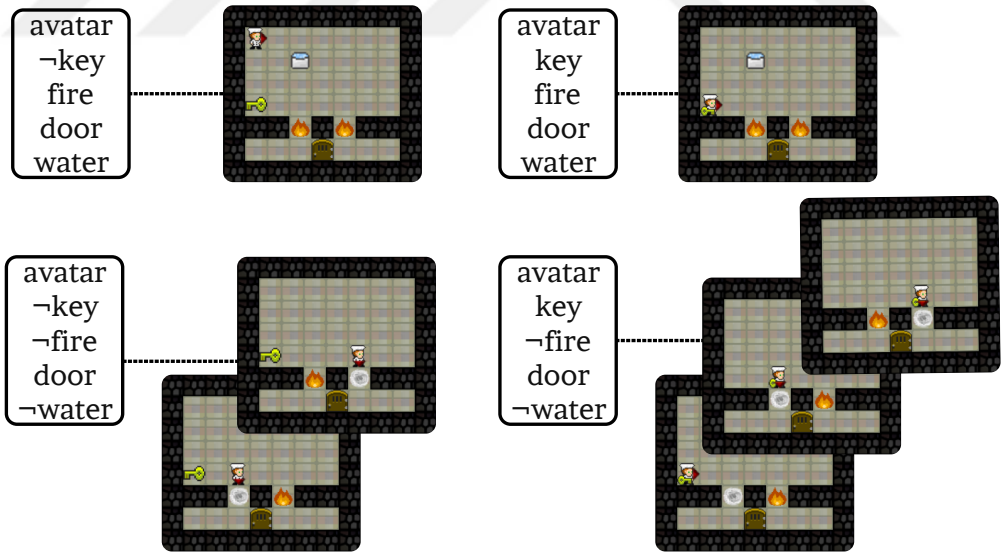


Figure 2: ©2019 IEEE.
Realized nodes of Game Scenario Graph from Figure 1.

interpreted by the GVG-AI game engine to simulate gameplay. Hence, we altered VGDL descriptions to insert faults into these games. These faults affect the game's implementation to behave differently

from the ideal design of the game. In order to create synthetic test goals, we sketched the scenario graph of these games and used this scenario graph and game sprites to generate synthetic test goals. Next, we used IRL to generate human-like test goals. However, we had to develop our IRL approach, MGP-IRL, to extract the tester behavior as test goals. Lastly, we used SARSA and MCTS RL agents to realize these test goals. Our experiments showed that agents provide comparable testing experiences to that of humans. Additionally, we compared the bug-finding performance with the bug-finding performance of game playing agent to constitute a baseline. Testing agents' bug-finding performance was superior to regular game-playing agents.

On the other hand, we also observed that SARSA could find more bugs than MCTS. However, we also observed that due to the stochastic nature of MCTS, MCTS was able to discover different bugs in various executions. In GVG-AI, several enhancements [68, 25, 87, 102] are employed to increase the performance of the Vanilla MCTS. Moreover, the authors [25, 87] noted that not every enhancement has an equal contribution to the performance. Furthermore, in board games, different MCTS enhancements [74] were favored in different games. Therefore, in "Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing," we experimented with several MCTS modifications and compared them under two distinct computational budgets [4]. We aimed to analyze their impact on the bug-finding performances of our agents. We experimented with six enhancements in this regard, and we introduced a new tree reuse strategy within these enhancements. Our experiments showed that MCTS agent that realize human-like test goals could surpass SARSA agents and human testers in finding bugs. However, the MCTS agents that realize synthetic test goals found fewer bugs than the SARSA agents.

These experiments presented two essential principles for game testing agents. First, the capability to re-train the agent to find different bugs is essential. MCTS discovered a new trajectory to test even when executed with the same test goal. Second, synthetic test goals are better realized with potent RL agents such as SARSA[89], DQN[55], or PPO [83]. MCTS only plans within its computational budget; therefore, depending on the game's complexity, the MCTS agent might not have found a viable action. The importance of synthetic test goals is that these goals can be generated without any human test data. Therefore, synthetic agents can be employed earlier than human-like agents to test games. Nevertheless, the stochasticity of SARSA agents is very low compared to MCTS agents. Therefore, an advancement for RL agents toward game testing should be to improve the ability to find various paths. RL algorithms such as Deep Q-Network (DQN) [55], Proximal Policy Optimization (PPO) [83], and Monte Carlo Tree Search (MCTS) [14] disregard the previous trajectories. Consequently, even if we train an agent with any of these algorithms and then evaluate the agent and repeat this process numerous times, all the generated trajectories would be similar. However, the trajectories may be different due to the following reasons a) the random initialization of the Neural Network (DQN and PPO), b) ϵ-greedy policy of DQN, c) stochasticity of MCTS, and d) the game's nondeterminism. The critical point is that even if the agent generates a distinct trajectory, this result is not by design but by random chance.

In order to diversify the paths, one has to change the feedback mechanism of the environment. Procedural personas [36, 56] accomplish this by rewiring the feedback mechanism with a utility function. An agent representing a persona will learn a different policy than another agent representing a different persona. However, the procedural persona approach also falls short when the game designer wants to see different playstyles within the same persona. For example, the game designer may want to see how other players complete a game with multiple endings. To model these players, she trains an agent that mimics the Exit persona and analyzes the trajectory of this agent's execution. Nevertheless, the

resultant trajectory of this persona will be the path to the closest ending. As a result, the other endings in the game will be neglected, and the game designer will only have to playtest data corresponding to one possible end of the game. A preliminary solution to this problem is masking the feedback from some endings. Thus, the agent will generate a playtest towards a particular ending. However, this solution requires additional tinkering, and there might be additional playtests towards the same ending. Another subpar solution is that the game designer would apply randomness to the agent's actions or add random noise to the input to diversify the trajectories. However, randomness does not guarantee that the agent will generate different playtests. Therefore, this solution also does not give complete control to the game designer.

On the other hand, with human playtesters, the game designer could have asked a playtester to play differently. The playtester already knows which paths or particular states she has visited before, so she uses this past knowledge to play the game differently. Therefore, the source of this problem is that the current agent does not know what the previous agents did in the previous runs. Therefore, every playtester which an RL agent represents generates a playtest anew. In order to solve this problem, we proposed Alternative Path Finder (APF) [6] in "Playtesting: What is Beyond Personas."

We use the exploration methodology to provide a basis for APF. Exploration methods in RL improve the agent's policy by motivating the agent to explore the environment. As the agent explores an environment, the agent improves its policy. The researchers proposed methods to encourage the agent to explore less visited states [11, 26, 66]. Compared to the traditional exploration methods such as ϵ-greedy, where exploration is achieved through randomness, these modern algorithms entice exploration logically. These algorithms learn to distinguish the unvisited states from the visited states; consequently, these algorithms guide the agents to less-visited states. As a result, exploration methods vastly improved the agent's score, such as Montezuma's Revenge [11]. As in exploration, APF also knows the previous trajectories. However, there is a clear distinction between exploration and APF. Exploration methods aim to increase the agent's knowledge about its environment during training. Therefore, this agent delivers top performance in this environment when we evaluate. The goal of APF is to help the agent to discover different performances without changing the agent's goal. Therefore during training, APF modulates the reward structure so that the old performances are penalized, and different performances are rewarded.

APF knows the previous trajectories and guides the agent to learn to play differently from previous ones. For this purpose, APF penalizes the agent when the agent visits a similar state and rewards the agent when the agent visits a different state, compared to the states in the previous trajectories. APF employs the state comparison algorithms used in exploration algorithms. These state comparison algorithms are the backbone of exploration research, and researchers tested these algorithms in multiple games. We show how we build the APF framework to generate new and unique playtests and how APF augments an RL agent.

Additionally, the playtesting process may employ players with distinct playstyles. These players will respond to the game differently and generate different play traces. The game designer can use these play traces to shape her game. In order to automate playtesting with different players, researchers replaced these playtesters with procedural personas. A procedural persona describes an archetypal player's behavior. Researchers used personas to playtest a Role-Playing Game [36] and a Match-3 [56] game. As a result, personas enabled distinct playstyles and helped to playtest a game like distinct players.

In order to realize the personas using RL agents, researchers used a utility function [38] to define the decision model of a persona. This utility function was used as the reward function of the Q-Learning agents. However, this replacement makes the agents bound to the utility function. Since the utility function is tailored for a specific decision model, the behavior of these agents is constant throughout the game. Therefore, the procedural personas approach is not flexible enough to create personas with developing decision models. For example, a player may change her objectives while playing the game. Consequently, the decision model of this player cannot be captured by a utility function.

Bartle [9] presents these changes a player can undergo while playing a Massively Multiplayer Online Role-Playing Game. We think that the change in the playstyle occurs after accomplishing a goal. For example, a player may start a game by opening the treasures to find a required item and then killing monsters. This player chooses her actions like a Treasure Collector until she finds the desired item and becomes a Monster Killer. We propose a sequence of goals to model the decision-making mechanism of this player. The sequence-based approach was previously used in automated video game testing agents [5] and was more practical than non-sequence-based approaches.

We proposed the developing persona [6] to overcome the shortcomings of procedural persona in "Playtesting: What is Beyond Personas." The developing persona model consists of multiple goals that are linked. Each goal consists of criteria and a utility function. The utility function serves the same purpose as in procedural personas. The criteria determine until which condition the current goal is active. When the current goal criteria are fulfilled, the next goal becomes active. The agent plays until the last goal criterion is fulfilled or until the end of the game. The game designer sets the criteria and utility functions of each goal. The goal structure enables the creation of dynamic personas. Additionally, this approach gives a more granularized control over a persona. The game designer can create variations of Monster Killer by setting different criteria. In order to playtest a casual Monster Killer, the game designer may set a health threshold as the criterion; and to playtest a hardcore Monster Killer, the game designer may set the percentage of monsters killed as the criterion.

1.1 Research Questions

In this thesis, we ask the following questions regarding the test goal generation. We propose two different methods, synthetic and human-like, and we use two different RL agents to realize these test goals Sarsa and MCTS.

- Which test goal technique is better for finding bugs?
- What is the difference between MCTS and Sarsa agents in bug finding?
- Which human-like agent is more similar to human testers?

Next, we enhance MCTS with several modifications and investigate these modifications regarding the computational budget. We ask the following research questions to analyze the effect on bug-finding.

- What is the impact of different computational budgets?
- Which modifications enhance MCTS's bug finding performance?
- What is the bug finding performances compared to Sarsa(λ) that uses the same test goals?
- What is the effect of modifications on human-like behavior?

Furthermore, we consider the following research questions on Developing persona when comparing the Procedural personas.

- How does a goal-based persona perform compared to a procedural persona?
  - Diversity of playtests generated by personas
  - Agreement between interactions performed and Persona's decision model

Lastly, we examine the Alternative Path Finder with the following research questions.

- Which additional paths can be discovered with APF?
- Does APF guarantee whether the agent find a different path?

1.2 Contributions of the Study

We list the contributions of this thesis as follows.

- Human-like tester agents.
- Synthetic tester agents.
- A test state to capture tester strategies and play them.
- Analysis of MCTS modifications for game testing.
- A more flexible and potent persona model, developing persona.
- A way to discover alternative paths in games, Alternative Path Finder.

We published the following papers from this study:

- S. Ariyurek, A. Betin-Can and E. Surer, "Automated Video Game Testing Using Synthetic and Humanlike Agents," in *IEEE Transactions on Games*, vol. 13, no. 1, pp. 50-67, March 2021, doi: 10.1109/TG.2019.2947597.
- S. Ariyurek, A. Betin-Can and E. Surer, "Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing," 2020 *IEEE Conference on Games (CoG)*, 2020, pp. 25-32, doi: 10.1109/CoG47356.2020.9231670.
- S. Ariyurek, E. Surer and A. Betin-Can, "Playtesting: What is Beyond Personas," in *IEEE Transactions on Games*, doi: 10.1109/TG.2022.3165882.

1.3 Organization of the Thesis

This thesis is organized as follows: Chapter 2 gives preliminary information about Graph Testing, RL, MCTS, GVG-AI, and VizDoom. Chapter 3 describes the examples and methodologies of related research. We grouped the related research into the following sections Game Testing, Game Playing, Learning from Humans, MCTS Modifications, Playtesting, Personas in Playtesting, Automated Playtesting, and Exploration Methods in Reinforcement Learning. We present our methodologies in five Chapters, from 4 to Chapter 8. These methodologies are Test State, Test Goal Generation, MCTS Modifications, Alternative Path Finding, and Developing Persona. Next, we describe our experimentation setup in Chapter 9. We present the results of our experiments in Chapter 10. Chapter 11 discusses the outcomes of experiments regarding our research questions and contributions and limitations of the proposed methodologies. Lastly, we conclude this thesis with Chapter 12.

CHAPTER 2

PRELIMINARIES

We present the preliminary material in the following subsections: Reinforcement Learning, Monte Carlo Tree Search, VizDoom, and Graph Testing.

2.1 Reinforcement Learning

Supervised learning, unsupervised learning, and reinforcement learning (RL) are the three fundamental paradigms of machine learning. RL concerns how agents should act in an environment such that the actions that the agent takes maximize the cumulative reward [89]. An agent experiences an environment by observing and interacting with the environment. As a result of these interactions the environment changes constantly. This interplay between the environment and the agent could be formulated using a Markov Decision Process (MDP).

Markov Decision Process is a tuple (S, A, T, R) where S is the set of states, A is the set of actions, $T : S \times A \times S \to [0, 1]$ is the state transition probability matrix, and $R : S \times A \to \mathbb{Q}$ is the *reward function*.

The objective of an RL agent is to find the best action given a state. The best action corresponds to the action that will lead to the maximum cumulative reward. The policy of RL agent defines which action to take given a state. Consequently, through numerous interactions with the environment, RL algorithms improve the policy. Policy maps all the available actions in a state to a probability.

$$\begin{aligned} \delta &\leftarrow R(s, a) + \gamma Q(s', a') - Q(s, a) \\ Q(s, a) &\leftarrow Q(s, a) + \alpha\delta \end{aligned} \tag{1}$$

State-action-reward-state-action (Sarsa) is a model-free on-policy temporal difference RL method. In (1), $Q(s, a)$ —the *Q-function*— calculates the total accumulated reward an agent will take by acting with action a in state s. $R(s, a)$ represents the *reward function* and the parameters of this function are $\alpha \in [0, 1]$ is the learning rate, δ is the temporal difference error, and $\gamma \in [0, 1]$ is the discount rate.

The training is performed as follows, the agent starts interacting in this environment according to a policy, and this interplay continues until the environment terminates. The termination of the environment could occur due to a success, a failure, or a time limit.

In the case of Sarsa, for each episode, the *Q-function* is iterated starting from an initial state. Sarsa is a temporal difference learning (TDL) algorithm. TDL algorithms use the previous estimate of $Q(s, a)$ and the observed reward to update the new estimate $Q'(s, a)$. Therefore, TDL algorithms do not require T, the state transition probability, of an environment, which makes Sarsa model-free. The $Q(s, a)$ is updated using the $a' \in A$ selected from the current policy which leads to an on-policy algorithm.

Eligibility traces is an extension to TDL where the $Q(s, a)$ is updated using the reward collected in subsequent states as well. In contrast, Sarsa only updates the $Q(s, a)$ using the reward collected by taking action a at the state s. Consequently, eligibility traces speeds up the learning drastically. While calculating the $Q(s, a)$ update, the rewards collected in the subsequent states are decayed by λ. In this study, we use Sarsa(λ) to refer to the Sarsa that uses eligibility traces.

Lastly, the *Q-function* can be represented using a table of state and action pairs (tabular)[89], or by a function approximator such as a neural network[55, 83]. Furthermore, there is the dilemma of exploration/exploitation. Exploration gathers more knowledge, and exploitation chooses the best action with the current knowledge. The agent's objective is to maximize the total expected reward; therefore, it must balance exploration/exploitation during training.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [14] is a search algorithm that builds a tree to get the best available action. MCTS consists of four consecutive steps: selection, expansion, simulation, and backpropagation. These steps are executed iteratively until a computational budget expires. A computational budget is a constraint on a resource such as time or number of iterations. When the computational budget expires, a child of the root —the best available action— is returned.

The Monte Carlo Tree Search [14] (MCTS) algorithm is a search technique that constructs a tree to find the best possible action. The four sequential phases of MCTS are selection, expansion, simulation, and backpropagation. These steps are carried out continuously until a computational budget expires. A computational budget is a limit on a resource, such as time or the number of iterations. When the resource allotted for computing resources has been used up, a subtree of the root —the best possible action— is returned.

$$UCB1 = \tilde{X}_i + 2C_p\sqrt{\frac{2\ln n}{n_i}} \tag{2}$$

The selection step selects a node using a Tree Policy. Upper Confidence Bounds (UCB1) shown in Equation 2 is a well-known technique for this policy. $\tilde{X}_i$ is the average score of the i^{th} child, C_p is the exploration constant, n represents the visitation amount of the root, and n_i is the visitation count of the i^{th} child.

The search tree is expanded during the expansion phase by adding one of the chosen node's unvisited children. Then, a simulation is rolled out from this unvisited child to estimate the child's value. A default policy is used in simulation to produce actions. This strategy produces completely random actions. The agent acts according to these actions and the reward obtained through this simulation is back propagated from this child to the root.

2.3 GVG-AI

GVG-AI [69] is a framework that includes various 2D games. There are almost 120 single-player games available, including well-known titles such as Mario, Sokoban, and Zelda. GVG-AI offers a demanding and engaging environment due to the diversity of games. GVG-AI games have another unique feature: they are all written in Video Game Description Language (VGDL) [82]. This language specifies the game rules for a particular game, such as what happens if the avatar strikes an adversary or interacts with a key.

In this thesis, we look at the faults between game implementation and game design. The GVG-AI framework builds a game by processing VGDL source code and running it through the GVG-AI engine. We are not evaluating the internal GVG-AI engine but rather the produced game. Thus, we consider VGDL as the implementation.

2.4 VizDoom

The VizDoom [44] is a platform that wraps the infamous Doom game as an RL environment. The VizDoom offers a flexible, efficient, and lightweight RL training platform. The environment is a 3D world, where the agent sees the environment from a first-person perspective, and the agent's action space is much larger than GVG-AI. Furthermore, the agent only observes a part of the game state, which adds additional complexity to RL training.

2.5 Graph Testing

There are various systematic approaches to generate tests and assess the suitability of the test set. First, testers model the system being tested (SUT) and use this model to identify defects. A standard way to model software is using graphs. Using graphs is standard practice to model software. After a graph is created to model the SUT, one can employ systematic testing methodologies to develop test scenarios.

A directed graph is a structure that consists of nodes N and edges E where $E \subseteq N \times N$. A path sampled from this graph is represented as a series of nodes $[n_1, n_2, ..., n_M]$, where each pair of consecutive nodes is contained in the set of edges E. In our study, the graph corresponds to the game scenario graph given by the game designer. Furthermore, depending on the graph coverage criterion, this graph may be used to produce numerous paths. These paths correspond to different ways to play the game. In this thesis, we utilized the all-path coverage criterion. All-path coverage criterion is met when the test set exercises each path in the graph [3].

CHAPTER 3

LITERATURE REVIEW

3.1 Game testing

Software testing is a dynamic investigation for validating the software quality attributes. In the case of game software testing, these attributes include cross-platform operability, aesthetics, performance in terms of time and memory, consistency, and functional correctness in a multi-user environment [80].

In order to validate these attributes, researchers proposed various methods to generate test sequences: record/replay [64], handcrafted scenarios [19], UML and state machines [40], Petri nets [34], and RL [71, 51]. However, when the game environment changes, test sequences and scenarios such as [64, 19] become obsolete, and a manual tester effort is required to create a new test sequence. Unlike these manual techniques, UML-based techniques automate the test generation process. However, generating sequences from UML runs into state explosion for larger games; without a gameplay AI, it relies on the generated states to play the game. Hence, researchers employed AI to test games. A Petri net that contains high-level actions of a game is used to generate test sequences using an AI [34]. RL is used with short and long-term memory to test Point-and-Click games [71]. Furthermore, RL is used to test the crafting system in Minecraft [51].

More generalized approaches to test video games are used to find bugs in two GVG-AI games [52]. As the authors [52] have stated, since the agent was a gameplaying agent, the agent was not required to find all the bugs. Lastly, a team of agents with different purposes is introduced to test games [32]. Nonetheless, all of these studies lack controlled experimentation.

After the initial success of Deep Reinforcement Learning, researchers explored the viability of generating a game testing agent. The agent mentioned in the studies discussed above was mainly a gameplaying agent, unavailable to find bugs outside the intended game paths. Wuji [100] generates test goals using evolutionary algorithms where the fitness is to explore game states, and the RL agent learns to play this test goal. The authors execute an automated test oracle during RL agent training to detect bugs. Nevertheless, there is no guarantee that evolutionary algorithms will generate goals that span all game sequences. Automatic video game exploration is beneficial in Super Mario World and The Legend of Zelda [17]. The authors augmented the collected trajectories with an exploration algorithm to increase the state coverage. Exploring the environment [30] to improve the state coverage is utilized to automate game testing in 3D environments. Though exploration helps increase state coverage, this approach does not guarantee to test a particular set of scenarios. In order to assert that specific scenarios are tested, researchers [65] combined Behavior-driven development with writing test scenarios.

On the other hand, an automated test oracle has uttermost priority for automated game testing. GLIB is an NN trained to detect UI and visual glitches for video games. However, this method cannot detect bugs where the visual is correct, but the action that led to this visual was faulty. In conclusion, automated game testing still has a long way to go [72].

3.2 Game Playing

Researchers have used RL and MCTS in various games, and the literature on these topics is rich. For instance, Sarsa(λ) is used to construct a human-like agent in Unreal Tournament [28] and as a game-playing agent in Ms. Pac-Man [94]. Afterward, tabular RL methods were traded with function approximators.

DQN [55] demonstrated that RL agents could surpass humans in several Atari games. Additionally, RL architectures that intermix MCTS and NN and a custom learning plan, AlphaGo [85], proved that agents could beat humans. Similarly, AlphaStar [96] which uses a custom architecture and a learning plan, was able to beat top StarCraft players. Last but not least, OpenAI Five [60] also uses a custom architecture and a learning curriculum to train agents in Dota2.

On the other hand, in general game playing, MCTS's aheuristic and anytime characteristics are prevalent. In order to increase the performance of vanilla MCTS, researchers proposed several modifications [25, 87]. Amongst them, knowledge-based evaluation (KBE) [68] is found beneficial. Another highly used algorithm in general game playing is Rolling Horizon Evolutionary Algorithms (RHEA) [27]. However, like MCTS, RHEA requires modifications to become competitive in a game, and there is no guarantee that a modification would work in another game.

Lastly, there are studies for general game playing [29] and MOBA [99] games that use Deep Reinforcement Learning. Although these papers aimed to create better agents in game playing, our purpose is to create an agent that tests the game by playing with respect to test goals.

3.3 Learning From Humans

Although it is crucial to include domain expertise, determining the right set of rewards is challenging. Furthermore, even humans learn better when coached by or imitated by an expert. Inverse reinforcement learning (IRL) is the study of extracting a reward function given an environment and observed behavior sampled from an optimal policy [59]. In the literature, different IRL approaches exist to learn the reward function. First, some approaches assume that the trajectories are sampled from the same policy [1, 101]. Next, some approaches assume trajectories could be sampled from different policies [7]. Then some approaches assume many sub-goals exist within the same trajectory [54, 88]. Consequently, given a trajectory, each method may extract a different set of reward functions.

On the other hand, testers may apply different strategies during their playtest. Consequently, their trajectories fit better with the multiple sub-goals methods [54, 88]. However, as noted by [88, 54] fails to generalize to unseen states, and the approach in [88] finds sub-goals only at the same level. Nevertheless, we aim to apply the learned test goals to other levels. Thus, we proposed MGP-IRL to overcome these problems [5].

Last but not least, human-like agents are employed as game playing agents in Unreal Tournament [92], Super Mario [61], Catan [23], GVG-AI [45], Spades [22], and Candy Crush [31].

3.4 MCTS Modifications

MCTS modifications are major upgrades to vanilla MCTS. Consequently, researchers employed various modifications to improve the performance of MCTS agents. For example, in GVG-AI, researchers experimented with various modifications [68, 25, 87, 102] and showed the advantages of these modifications. However, as noted by authors [25, 87], the contribution of each modification is not the same and may change depending on the game. Furthermore, different modifications were found favorable in board games [74]. On the other hand, we want to improve our MCTS agent's bug-finding performance. Nevertheless, the research on this topic is blank; thus, we list MCTS modifications for game-playing research.

MCTS represents and explores the game as a tree. This tree may have several nodes corresponding to the same game state. Consequently, a piece of critical information stored in one of these nodes is not transmitted to the other nodes. In order to tackle this issue, transposition tables (TT) were introduced to MCTS by Childs et al. [18]. TT encourages information exchange across nodes in the MCTS tree that correspond to the same game state. The authors utilized this shared information to determine the UCB1 score of a node, and they provided three techniques to calculate this score. TT is used in games such as Deep Sea Treasure [70], General Game Playing [103] (GGP), and Hearthstone [20]. Lastly, Xiao et al. [98] used a state's feature representation to search for related states in memory and enhanced knowledge distribution.

It is often challenging to locate a terminal state or even a state that alters the game score in GVG-AI, which may lead the MCTS agent to act erratically. Powley et al. [74] presented modifications to capitalize on games' episodic nature. Their modifications have proven valuable in games such as Dou Di Zhu, Hearts, and Othello. Soemers et al. [87] experimented with eight modifications, including KBE. Their experiments revealed that in GVG-AI, KBE modification was the most prominent. Also, in GVG-AI, İlhan and Etaner-Uyar [102] used temporal difference learning to extract domain knowledge from past experiences and used this knowledge to boost their MCTS agent. Finally, Silver et al. [85] trained a value NN to analyze the game state in Go successfully, and AlphaGo defeated the world champion in Go using this NN.

The tree reuse strategy uses the previously generated tree to guide the forthcoming MCTS runs. Moreover, pruning this tree is as simple as selecting the subtree of the selected child. Nevertheless, using the previous tree as is may increase the memory requirements and put additional constraints on the computational budget, which are essential concerns for MCTS. In Ms. Pac-Man, Pepels et al. [67] suggested a decaying tree reuse approach. This decaying tree reuse approach was utilized by Soemers et al. [87] in GVG-AI games and by Santos et al. in Hearthstone [79].

MixMax modification is introduced by Jacobsen et al. [41] to counter the cowardice behavior of the MCTS agent in Super Mario Bros. In GVG-AI, MixMax is used by Khalifa et al. [45] to create a more human-like MCTS agent. Furthermore, Frydenberg et al. [25] used MixMax in GVG-AI. However, their findings for MixMax were mixed.

The simulation policy in Vanilla MCTS chooses random actions in rollouts. Finnsson and Bjornsson [24] calculated the likelihood of actions in General Game Playing (GGP) using Gibbs sampling. As a result, the authors skewed the simulation policy by choosing probability-based action. According to Tak et al. [90], this simulation policy does not set the selection probability of the optimal action. As a result, they employed ϵ-greedy to improve this simulation policy. Additionally, Powley et al. [74] compared Gibbs sampling and ϵ-greedy approaches in GGP and found that ϵ-greedy is preferable. Silver et al. [86] employed softmax to parameterize the simulation policy in Go. Perez et al. [68] used the learned experience to bias rollouts in GVG-AI. In contrast, James et al. [42] examined why more informed rollouts result in less effective agents. According to their research, knowledge-based rollouts may lead to high bias and low variation, leading to poor performance.

3.5 Playtesting

Playtesting is a technique used in the game development process. Playtesters test a game, and their feedback is gathered. Afterward, game designers analyze and study this feedback and then improve their game. Since playtesting requires a human effort, researchers suggested strategies to automate playtesting. First, Powley et al. [73] combined automated playtesting with a game creation tool. Next, Gudmundsson et al. [31] built an NN architecture to predict the most human-like action in Candy Crush. Afterward, this architecture measures the difficulty of Candy Crush levels. Finally, Roohi et al. [78] utilized RL to predict Angry Birds Dream Blast's level difficulty. The automated playtesters in these experiments are considered to belong to the same player archetype.

On the other hand, during a playtest, each playtester may belong to a different player archetype. For example, one player may want to enjoy the game as much as possible, while another may want to see how fast she can finish the game. Consequently, depending on the application, the data collected from these two playtesters should be used differently.

3.6 Personas in Playtesting

Personas are used in playtesting to give game designers an idea of how various players would approach the game. These unique player types are represented by Personas, which are fictitious characters. For example, Bartle [10] identified four unique personas in a Multi-User Dungeon Game Socializers, Explorers, Achievers, and Killers. Furthermore, he laid the groundwork for these four personas' interests and typical acting patterns. Afterward, Bartle [9] expanded on this study by providing persona development sequences. The development sequences explain how and why a player may shift to a different persona.

Furthermore, Tychsen and Canossa [93] collected game metrics from the game Hitman Blood Assassin. The authors used this collected data to classify the human playtesters into *Mass Murderer*, *Silent Assassin*, *Mad Butcher*, and *The Cleaner*.

To sum up, these researches focused on identifying which persona is playing the game and what would these personas would do in this game.

3.7 Automated Playtesting

Researchers suggested using RL agents to realize a persona's decision model to automate playtesting. Holmgård et al. [38] employed a utility function to implement the decision model of a persona. For each persona, the authors created a different utility function. Later, this utility function replaced the reward function of the Q-Learning agent. Consequently, they trained a different RL agent for each persona. The authors also crafted an environment to evaluate the RL, MiniDungeons. The results showed that each RL agent generated a different play trace as a persona would. Holmgård et al. [37] expanded on their prior work using a neural network instead of Q-Learning. The authors handpicked the neural network's inputs based on heuristics. The weights of this neural network were determined using a genetic algorithm. Hence the authors named their new agent as "evolved agent." Their experiments revealed that —compared to the Q-Learning agent— the evolved agent needed less training and was better at generalizing to other levels.

Building upon their work, Holmgård et al. [39] proposed using an MCTS agent for personas. Their motivation for employing MCTS [14] was to expedite the playtesting process. In contrast to the Q-Learning and evolved agent, MCTS required no training time. Furthermore, they conducted their experiments in a new environment called MiniDungeons 2. Though MCTS agents provided playtest data quickly, the agent's success decreased. Finally, Holmgård et al. [36] improved upon their previous work by applying modifications to the Vanilla MCTS. Their proposed modification improved the selection phase of MCTS. Furthermore, the authors noted that this new environment was too difficult for Vanilla MCTS. Therefore, the authors substituted the UCB1 equation with a new formula. A genetic algorithm constructed this new formula, and each persona formed a different formula. Their experiments demonstrated that this modification is an essential improvement and the results matched with the correct persona model.

For Match-3 games, Mugrai et al. [56] handcrafted four distinct personas, which are *Max Score*, *Min Score*, *Max Moves*, and *Min Moves*. The authors used MCTS to realize these personas. The experiments demonstrated that these four personas provide useful information to the game designer. On the other hand, besides RL agents, Silva et al. [21] designed a set of heuristics that represents a unique persona in the Ticket to Ride board game. These personas were selected amongst the competitive personas and played against each other. Their experiments revealed unique encounters and situations where the game rules were lacking.

The utility function is the fundamental disadvantage of persona research. First, the utility function is constant and static throughout the game. As a result, the development sequences defined by Bartle [9] cannot be realized. Additionally, the personas may generate a similar playtrace depending on the level arrangement [36]. These issues may result in synthetic playtesters producing inadequate feedback to the game designer. Lastly, synthetic playtesters may not playtest all playable pathways since RL agents maximize the overall cumulative reward.

3.8 Exploration Methods in Reinforcement Learning

An RL agent interacts with the environment to determine which action offers the maximum reward in a given state. Consequently, the RL agent must explore this environment to improve this policy. The

exploration problem is how to motivate an RL agent to explore the unexplored part of an environment. The researchers presented various methods to tackle this problem.

Count-based techniques promote rarely visited states over frequently visited ones. As a result, the agent is rewarded for exploring these rarely visited states. Researches experimented with the following methods to formulate a count for the states, a density model [11], a neural density model [63], a hash table [91], and exemplar models [26]. The uncertainty measurement of the agent is a different strategy to provide an extra reward to entice exploration. Researchers formulated uncertainty using bootstrapped DQN [62], state-space features [66], and neural function error [15]. Furthermore, researchers presented methods for exploring the state space by optimizing the state marginal distribution to match a target distribution [48].

Last but not least, the exploration methods aim to encourage the agent to discover the environment and not to find different ways to play. Consequently, the agent will learn to play the game as best as possible.

CHAPTER 4

METHODOLOGY I: TEST STATE

Testers evaluate a game using various approaches [76], their approaches lead them, and these testers interact with different game aspects. However, when the same tester re-evaluates the same game, the tester interacts with aspects different than before. Because the tester has already tested a particular portion of a game and now tests another portion. This behavior of testers' is akin to a path-finding algorithm, such as Depth First Search (DFS). DFS retains which nodes it has visited, which prevents the DFS from terminating. Similarly, these testers mark specific interactions as "tested" in their memory, in software, or on a piece of paper. This "memory" prevents testers from performing already tested interactions.

An MDP may be used to model interactions that advance the game, such as picking up a key. On the other hand, modeling activities that do not progress the game using only the game state, such as attempting to get through a wall, is challenging. If the game does not permit this interaction, the actor should remain in the same position. Additionally, an actor may repeatedly strike the wall if a positive reward is received from this interaction. We could solve this state representation problem by labeling this wall as "tested" in a memory. This interaction is not recorded in the game state but memory. As a result, when we combine the game state with a memory, the MDP formulation becomes simpler.

Pfau et al. [71] use a concept similar to memory to explore an adventure game. This memory is then used to modulate the reward of each available action. However, their usage of memory was limited to Point-and-Click games, and we propose a memory to be used for 2D grid games. Therefore, we propose a grid-based memory model and refer to this memory as the test state. This test state marks the interactions executed by the human testers of RL agents. We use this test state as a supplement to the game state. In game testing, if we utilize the game state without the test state, then our MDP will be insufficient. First, the game state can only be altered via specific VGDL game rules, meaning that some interactions will have no effect. Second, because of a bug, even interactions that should change the state of the game may not change the state.

In games, interactions occur between entities. For 2D grid games, these interactions are between game sprites. We define a sprite as η and the set of sprites as Γ. Next, we refer to the two sprites as η_0 and η_1. Additionally, we can add more parameters to this interaction for game testing purposes. We use $Pos : <x, y>$ to indicate the position of the interaction and $Dir \in \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ to describe the direction of the interaction. The Dir refers to the direction of the first sprite η_0. However, if the first sprite does not have a direction, we use $\rightarrow$ as default. We use $Type$ to describe the action type. For GVG-AI games, we use the following $Type \in \{Move, Use\}$. Lastly, we use $Avatar_{State}$ to represent the types of an avatar in VGDL. To sum up, our interaction ζ definition is as follows $\zeta = < \eta_0, \eta_1,$

Pos, Dir, $Type$, $Avatar_{State}>$. We should note that our "interaction" definition differs from the interaction definition in GVG-AI.

Since we emphasize grid games, we utilize a test state constructed as a 3D grid. To cope with the following issues, we choose a 3D grid to represent the test state for a 2D grid game. First, the tester may prioritize interacting with a sprite from all directions or may prefer the same direction for all sprites. Second, the tester may perform an interaction several times. Lastly, the tester may perform her interactions using *Move* and *Use*. These issues are addressed with a 3D grid. Every layer in this 3D grid is a 2D grid with the dimensions of the game grid. We utilized 12 2D grids in the final implementation. The first four levels represent the *Move* interactions between the avatar and other sprites. The *Use* interactions between the avatar and other sprites are represented in the following four levels. The rest of the interactions are stored in the last four levels of this test state. For instance, when another sprite interacts with another sprite, such as other movable sprites, or when the avatar pushes the water bucket, and the water bucket moves as well. We used four levels for each of these groups to differentiate the direction of these interactions. Lastly, we store a count at each cell of this 3D grid. This count represents the total amount of interactions performed for a specific group, direction, and position.

Although we presented a test state for 2D grid games, the test state could also be used for different games. The primary concept is reproducing the visual environment and marking the tested interactions. In addition, other layers may be added to this test state based on the distinctions that must be made between various categories, such as movement and use. On the other, if the test scope is limited to a few sprites such as keys or doors. Then, we could use an array to verify if these sprites are tested or not.

For 2D grid games, we use the following definitions. A 2D grid game is a tuple $<\mathcal{G}, \mathcal{I}, A, \Gamma, \delta, ||\zeta||>$, where $\mathcal{G}$ is the set of game states, $\mathcal{I}$ is the set of test states, $A = \{\uparrow, \downarrow, \leftarrow, \rightarrow, Use, Nil\}$ is the set actions where Nil signifies taking no action. Γ denotes the set of sprites, and the transition function is denoted by δ, which generates the next game state, test state, and the interactions from the current game state, test state, and action. $||\zeta||$ is the set of all interactions. We extend the S definition from the MDP (see Section 2.1) as $S : \mathcal{G} \times \mathcal{I}$.

We use the following feature definition $\phi = <\eta_0, \eta_1, Weight, Method, Type, Rep, Avatar_{State}>$. $\eta_0, \eta_1 \in \Gamma$, and *Weight* represents the received reward; the direction preference of the tester is captured in *Method* parameter, where *Method* $\in \{Each, All\}$. We use *Each* to differentiate different directions and *All* to consider all directions to be the same. The type of the action is represented in *Type* $\in \{Move, Use\}$. *Rep* restricts how often the award may be acquired. Lastly, $Avatar_{State}$ denotes the state of the avatar.

We calculate the *reward function* as follows. When an actor acts in the game, interactions are generated. These interactions are matched with the features to calculate the reward of performing this action. The matching procedure is as follows first, the feature ϕ matching to the $\eta_0, \eta_1, Method, Avatar_{State}$ parameters of the interaction ζ is found. Next, the *Rep* parameter of ϕ is compared with the count stored in the test state. The η_0, Dir, and $Type$ of ζ are used to extract the count value. If the count $\leq$ *Rep*, the actor receives the reward in *Weight*, and then the count value in the test state is incremented. If the $Method$ parameter of ϕ is *Each*, we only increment the cell corresponding to the Dir. However, if the $Method$ parameter of ϕ is *All* then we increment all four cells. Nevertheless, an interaction may

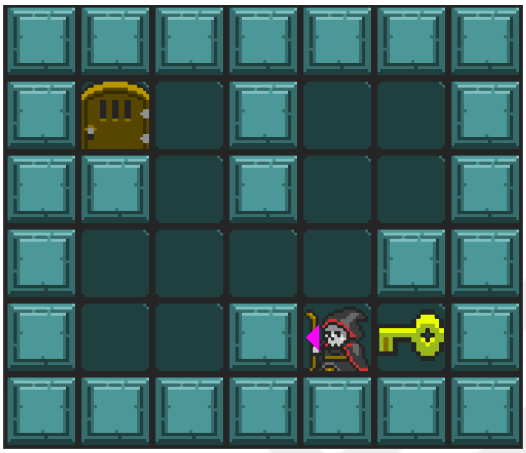
```
SpriteSet
  floor > Immovable img=oryx/floor3
  goal  > Door img=oryx/doorclosed1
  key   > Immovable img=oryx/key2
  sword > OrientedFlicker img=oryx/slash1
  movable >
    avatar  > ShootAvatar stype=sword
      nokey   > img=oryx/necromancer1
      withkey > img=oryx/necromancerkey1
  wall > Immovable img=oryx/wall3
InteractionSet
  movable wall > stepBack
  nokey goal   > stepBack
  goal withkey > killSprite
  nokey key    > transformTo stype=withkey
                   killSecond=True
```

Figure 3: ©2019 IEEE.
An example VDGL snippet with SpriteSet and InteractionSet.

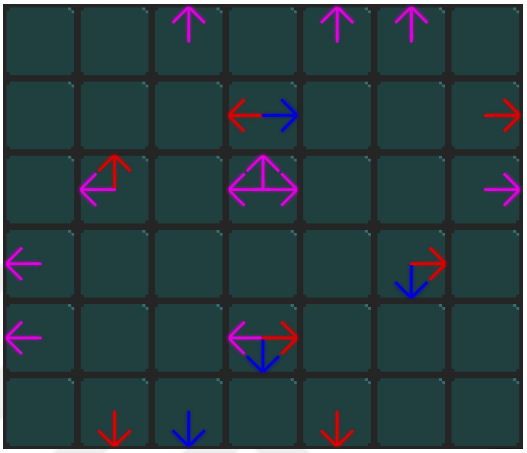
not match a feature, in this case, we return a predefined reward, and the test state remains unchanged. In our experiments, we set this predefined reward as -1.

Figure 3 portrays a game state, and three test states. The VGDL in Figure 3 denotes the rules of this game. The game state shown in Figure 4a contains the following sprites Γ=$\{Avatar, Door, Floor, Key, Wall\}$. A pink triangle shows the direction of *Avatar* and $Avatar_{State}$=$NoKey$. The two test states are shown in Figure 4b and Figure 4c, where the interactions in the first four levels are colored red, and the interactions in layers four through eight are colored blue. Hence, their combination produces the color magenta. The only difference between Figure 4b and Figure 4c is that the features of the former have *Each*, and the latter use *All* in the *Method* parameter. Furthermore, the arrows indicate state's stored interaction direction. The count values for the first four layers of Figure 4b are shown in Figure 4d. For convenience, the cells with zero counts are left blank.

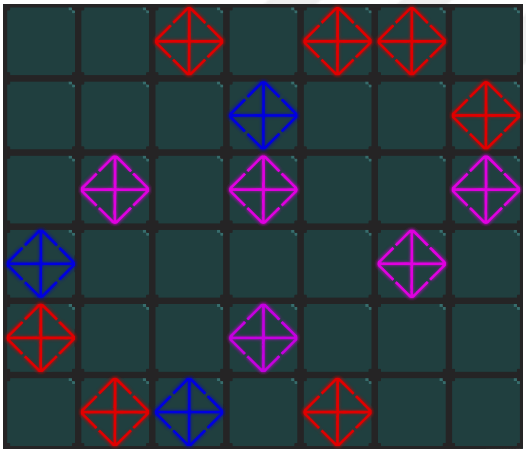
In the grid shown in Figure 4a, the top-left corner is $Pos : <0, 0>$ and the *Avatar* is at position $Pos : <4, 4>$. In Figure 4b, we see that the direction is $\uparrow$ at $Pos : <2, 0>$. Furthermore, at $Pos : <3, 4>$, we see that all directions are present besides $\uparrow$. Moreover, when the *Method* is *All*, we can no longer differentiate the direction (see Figure 4c). Lastly, by inspecting the colors in Figure 4b and Figure 4c, we can differentiate between the *Use* and *Move* interactions.



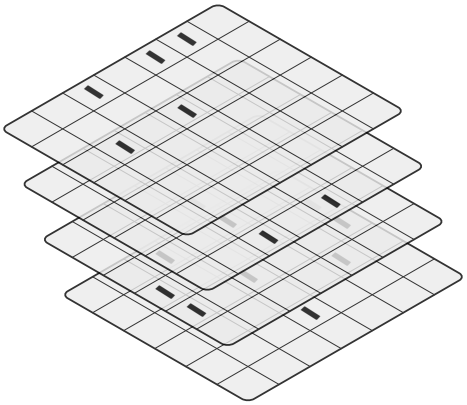
(a) Game State



(b) Test State with Method::Each



(c) Test State with Method::All



(d) Test State Layers of (b)

Figure 4: ©2019 IEEE.
Game State and Test State

CHAPTER 5

METHODOLOGY II: TEST GOAL GENERATION

5.1 Synthetic Test Goals

Testing is a crucial step in game development, and automation of this step is essential as it expedites development while lowering testing effort. However, even though strategies like playing a pre-defined or pre-captured scenario are beneficial, these methods become ineffective when the game layout changes. On the other hand, using RL agents in game testing offers more flexibility. Thus, in this Section, we describe a game scenario graph-based method to generate synthetic test objectives. RL agents use these test objectives and act as artificial testers.

Using graph coverage criteria, we generate several scenario paths using the game scenario graph. The game designer provides the game scenario graph, such as Figure 1. The game scenario graph in this example specifies permitted transitions. The paths generated from this graph cover different nodes of the game scenario. We aim to produce test objectives from these generated paths that an agent can comprehend. In this respect, we use features stored in the nodes to direct the RL agent toward the test objectives. As a result, when RL agents check whether every path generated from this game scenario graph could be executed, we can verify if the game scenario graph is implemented correctly.

In addition to checking if the game implements the scenario, we also want to check whether the game implements some other behavior that is **not** in the scenario, as any tester would do. These checks encourage the agent to ask: What happens if the avatar attacks the key? Can an avatar pass through these walls? In order to verify questions such as these, we create a list using all combinations of the following feature $<\eta_0$, η_1, $Type$, $Avatar_{State}>$. We refer to this list as the modifications list. Nevertheless, a game may have a high sprite count, and creating an all combinations list becomes enormous. We suggest using the pair-wise combinatorial [47] strategy in these cases. Last, we reduce the modification list by limiting the first parameter, η_0, to be movable.

The algorithm for creating synthetic test goals is as follows: The game designer provides a game scenario graph, the coverage criterion, and a map to match edges in this graph to abstract features. A feature is considered abstract when the *Weight*, *Method*, and *Rep* parameters are null. The algorithm then generates sequences in the form $[n_1, n_2, ..., n_M]$. First, this sequence is transformed by replacing each pair of nodes (n_i, n_j), the edge, with an abstract feature. We refer to this modified sequence as the feature sequence. The next step is to modify each of these created feature sequences by adding features from the modification list. This additional feature may be added at any point in the feature sequence. However, we restricted the modification process with the following requirement. We require the Hamming distance between the candidate feature from the modification list, and each feature in the

sequence must be at least one. The η_0, η_1, $Type$, $Avatar_{State}$ are the parameters of features used to determine the Hamming distance, which is the number of parameters where values are distinct. This requirement allows the feature sequence to be extended by different modifications. Since modifying with a modification whose Hamming distance is 0 would be meaningless.

Afterward, we prune our modification list. We need an approach to push these modifications to the feature sequences. Suppose we push all K modifications into a feature sequence with M features at once. In that case, we could insert up to $M \times K$ modifications —without taking the order of modifications into account. However, this testing method is inferior since one of the features from the modification list may crash the game, and we will not be able to test the other modifications. We propose to copy the entire feature sequence $M \times K$ times. Afterward, we insert a single distinct modification from the modification list to the cloned feature sequences. Lastly, we fill the missing parameters of abstract features as follows: if η_1 is a moveable sprite, the *Rep* parameter is set to 3; if η_1 is abundant in the level, such as walls, *Rep* is set to 1; otherwise, *Rep* is set to 2. The *Method* parameter is set to *All*, and the *Weight* parameter is set to 1.

Modifying the feature sequence will direct the RL agent to test various game mechanics. After modification of the feature sequence is done, we transform this modified feature sequence into a sequence of test goals. Compared to the goal state definition in [77, 54, 88], our goal state definition is different. The goal state is defined as S_h where $S_h \subseteq S$. However, if we want to represent the state S_h where we tested to get beyond every wall in the game, we could match multiple s to this goal S_h. Additionally, a test goal should provide the flexibility to check any amount of walls. However, the goal defined by [77, 54, 88] neglects this possibility. In order to overcome this problem, we also supplement each feature in the goal with a criterion. This criterion indicates the proportion of $eta1$ to be checked. As a result, the criteria determine if the agent achieved the test goal while the features direct the agent through the environment. Consequently, we define our test goal as h, and h is composed of features and criteria $\{(\phi_0, c_0), (\phi_1, c_1), ..., (\phi_n, c_n)\}$, where $c_0, ..., c_n$ represents the criterion for each feature. Finally, the sequence of test goals is represented as $\mathcal{H}$=$(h_0, ..., h_n)$. We set each criterion $c_0, ..., c_n$ as 100% for our synthetic test goals.

The RL agent is trained in a game starting with the first goal h_0. When the criteria are met, the agent moves to the next goal in the $\mathcal{H}$, in this case, h_1. The training continues until all of the $\mathcal{H}$ goals are processed. Nevertheless, the agent may not accomplish a test goal h for two reasons. First, the test goal can be impossible, like attacking an unreachable door. Furthermore, a bug may interfere, and the agent may get stuck. Consequently, we let the agent train in an environment for a certain number of steps since it is impossible to know if a test goal h is feasible. Therefore, we decided that if the agent cannot fulfill the test goal h in the given number of steps, the agent should not start the training with the subsequent test goal. However, in our implementation, we added the flexibility to let the game designer decide whether the agent should continue on training.

We accomplish three objectives with our sequence of test goals. First, we want the agent to execute the test goals in the desired order so that the agent traverses the scenario graph in the appropriate order. Second, the whole sequence of test goals may not be played due to a defect, but executing each test goal allows us to identify precisely where the error happened. Lastly, the sequence of test goals is a scalable approach. Since each goal corresponds to a node in the game scenario graph if the game scenario graph is large, only more test goals are added to the $\mathcal{H}$. We should note that the other approach was to flatten the sequence of test goals to a single test goal. However, we retained the sequence of goals approach [77].

Lastly, we add an exploration feature $<Avatar$, $Floor$, All, $Move$, 0.01, 1, $Avatar_{State}>$ to every test goal in $\mathcal{H}$. By adding this feature, we assist the exploration of the agent. While adding this feature, we copy the $Avatar_{State}$ parameter from the other features in the same test goal.

5.2 Human-Like Test Goals

The importance of beta testing cannot be neglected in game development. Human testers that participate in beta testing utilize their experience and heuristics to identify different issues. Furthermore, the activities of each participant may be logged for future use. These logs can be stored as trajectories, and a trajectory τ is a chain of actions $[a_0, ..., a_n]$ where $a_i \in A$ and $0 \leq i \leq n$, where n is the length of the trajectory. A trajectory does not indicate anything useful but reveals test goals when repeated in the same game environment. As a result, the gathered trajectories are employed in regression testing in the literature [64]. We propose to automate test goal creation and capture the knowledge of human testers by learning from their activities rather than copying or replaying them. In order to understand this knowledge, inverse reinforcement learning [89] (IRL) is used. IRL presumes that the trajectory is near-optimal. However, this assumption may not hold during ad hoc testing. Furthermore, the human tester could carry out a complicated series of tasks that are impossible to simulate using linear weights [54, 88]. To address these issues, we propose MGP-IRL.

MPG-IRL accepts that the trajectory may not be near-optimal but assumes that the trajectory could be split into multiple near-optimal chunks. MGP-IRL does this by aggressively splitting the trajectory and then merging these trajectories based on the likelihood threshold given by the user. When one trajectory can no longer be merged with another, MPG-IRL concludes that under the given likelihood threshold, this trajectory should be converted into a test goal. As a result, MGP-IRL generates a sequence of test goals $\mathcal{H}$ depending on the likelihood threshold.

Algorithm 1 describes MPG-IRL. The algorithm breaks the actions in the trajectory into the fewest possible trajectories and interactions in line 2. In order to split the trajectory, we iterate the trajectory from the beginning and split the trajectory where η_0, η_1, and $Type$ interaction parameters differ from the previous interaction. Line 3 initializes the previous likelihood threshold κ_a to zero, the goal sequence $mathcalH$ to empty sequence, the prior features set Φ_a to empty set, the previous trajectory set τ_a to empty sequence, and the previous features set Φ_a to empty set.

We iterate each of the divided trajectories one by one. Starting from the first segment, we transform this segment into a feature ϕ_i in line 6. The $Weight, Method$, and Rep parameters of ϕ_i are kept empty since they cannot be collected from interactions alone. We only create features for the observed interactions. Therefore, the IRL algorithm knows which features have non-zero weights [1]. This application of non-zero weights helps the IRL algorithm to learn using fewer trajectories. Then, in line 8 Φ_b is created by joining the feature ϕ_i with the preceding feature Φ_a. In line 9, we set the Rep parameter of the features in Φ_b. In order to find the Rep parameter, we replay the combined trajectory τ_b and analyze the repetitions by checking the interactions performed. Setting the Rep parameter is crucial for IRL to learn the $Weight$ of a feature correctly.

In line 10, we employ IRL to learn the $Weight$ parameter of the features Φ_b of trajectory τ_b. We used Maximum Likelihood Inverse Reinforcement Learning [7] (MLIRL) in our implementation since MLIRL calculates the likelihood of a trajectory and is robust to slight noise. Next, in line 11, the likelihood [7] of the trajectory κ_b is calculated using Φ_b. The probability of the trajectory κ_b is then

determined using Φ_b in line 11. We determine whether to merge trajectories and features based on this probability estimate. The trajectory τ_a may be sampled with the likelihood of κ_a from a policy π_a. However, we ask whether τ_i should be added to τ_a, sampled from the policy π_b with the likelihood of κ_b. To answer this question, we compare κ_b with κ_a. The difference κ_a - κ_b is negative, if π_b explains the trajectory τ_b better than π_a explains the trajectory τ_a.

In line 12, we check whether the difference between κ_b and κ_a is less than κ_T. κ_T is the likelihood threshold that is given as an input parameter to the algorithm. In our experiments, we investigated the impact of the threshold κ_T. We found that the agent could display the learned features more accurately if the threshold is 0. Furthermore, we could add that when the κ_T is larger, the algorithm becomes more analogous to the internal IRL algorithm. Afterward, if the criteria in line 12 are met, the previous features Φ_a is replaced with the current features Φ_b, the current likelihood κ_a is set as κ_b, and the current trajectory is set to τ_b. On the other hand, if we cannot fulfill these criteria, we transform the previous features Φ_a into a goal h. Furthermore, we calculate a criterion c for each feature in the Φ_a if the $Reward$ of this feature is non-negative.

We calculate the criteria for the goal h as follows. First, we repeat the trajectory τ_a and check the interactions. From the interactions, we count each feature in Φ_a. We represent this count as $countF(\phi_i)$ where $\phi_i \in \Phi_i$. Afterward, we count how many times η_1 appears in the game, $countS(\eta_1)$. For instance, in Figure 4a, the counts for $countS(Wall)$=27 and $countS(Door)$=1. We calculate the criterion for each feature in Φ_i as $\frac{countF(\phi_i)}{countS(\eta_1)} \times 100$. In order to achieve similar interactions in other levels, we normalized the $countF(\phi_i)$ by $countS(\eta_1)$. After we calculate the criteria, we progress the game $\mathcal{G}$ by applying actions from the trajectory τ_a, and then we reset the test state.

The algorithm looks for a remaining segment at line 19; if there is one, we turn this segment into a goal. Last but not least, not every tester will test the same aspect in the same manner. As we mentioned in Chapter 4, the tester may have $Direction$ preference. Though this part is not shown in the algorithm, we repeat the lines from 6 to 11 by alternating the *Method* parameter of each feature ϕ_i. We choose the *Method* which yields a better likelihood.

To conclude, we split the trajectory into segments by checking where the interactions differ. We created features using the observed interactions, which allowed us to create non-zero weight features, calculate the repetition count, and find the direction preference. This approach's most important benefit was learning policies within a likelihood threshold. For example, consider a human executed the following actions in Figure 4a. First, she attempted to enter the door while attacking the walls along the way, and finally, she attacked the Key. This trajectory is broken down into interactions by MGP-IRL. The weights, technique, and repetition cap for walls are discovered in the first iteration. In the next iteration, the algorithm considers combining the initial trajectory where she attacked the walls with the trajectory where she attempted to enter the door. The likelihood of the combined trajectory increases since the tester attacked a particular portion of walls, which are on the route to the door. Consequently, MPG-IRL joins these two trajectories and features. The algorithm examines merging with the trajectory attacking the Key in the following iteration. Nevertheless, adding this trajectory will decrease the likelihood as the Key could have been attacked earlier in the trajectory. Consequently, MPG-IRL merges this trajectory depending on the κ_T value. If this criterion is not met, the first combined trajectory is transformed into a goal, and the rest is transformed into another goal. These two goals are added sequentially to the $\mathcal{H}$. On the other hand, if the criterion is met, the whole trajectory is transformed into a single goal and then added to $\mathcal{H}$.

Our technique differs from [54] and [88] in that their sub-goal definition is state-based, limiting the application of the goals to other levels. Furthermore, our feature definition also allows us to learn the number of repeats and direction preferences, which are not calculated in their technique.

Algorithm 1 ©2019 IEEE.
Multiple Greedy Policy Inverse Reinforcement Learning

procedure MGP-IRL(G, τ, κ_T)
  $\zeta_{0..n}, \tau_{0..n}, n \leftarrow$ SPLITTRAJECTORY(G, τ)
  $\Phi_a \leftarrow \{\}, \tau_a \leftarrow [\,], \kappa_a \leftarrow 0, \mathcal{H} \leftarrow [\,]$
  $i \leftarrow 0$
  while $i \leq n$ **do**
    $\phi_i \leftarrow$ CREATEFEATURE(ζ_i)
    $\tau_b \leftarrow$ CONCATENATE(τ_a, τ_i)
    $\Phi_b \leftarrow \Phi_a \cup \phi_i$
    $\Phi_b \leftarrow$ ANALYZEREPETITIONS(G, τ_b, Φ_b)
    $\Phi_b \leftarrow$ IRL(G, τ_b, Φ_b)
    $\kappa_b \leftarrow$ CALCULATELIKELIHOOD(G, τ_b, Φ_b)
    if Φ_a is $\{\}$ OR $\kappa_a - \kappa_b \leq \kappa_T$ OR Φ_a is Φ_b **then**
      $\Phi_a \leftarrow \Phi_b, \tau_a \leftarrow \tau_b, \kappa_a \leftarrow \kappa_b$
      $i \leftarrow i + 1$
    else
      $G, h \leftarrow$ CREATEGOAL(G, Φ_a, τ_a)
      $\mathcal{H} \leftarrow$ APPEND($\mathcal{H}$, h)
      $\Phi_a \leftarrow \{\}, \tau_a \leftarrow [\,], \kappa_a \leftarrow 0$
  if Φ_a is not $\{\}$ **then**
    $G, h \leftarrow$ CREATEGOAL(G, Φ_a, τ_a)
    $\mathcal{H} \leftarrow$ APPEND($\mathcal{H}$, h)
  return $\mathcal{H}$

5.3 Generating Test Sequences

Testing is a demanding process, and any change to the software or concept necessitates re-testing. Testing gets more and more tedious due to this necessity. Consequently, the capability to automatically generate new test sequences is crucial. Our solution uses RL agents to automatically generate new test sequences based on human-like or synthetic test goals. This section describes how our agents transform the test goals into test sequences.

Reinforcement learning algorithms require an environment to perform. This environment contains a reward vector. However, this reward vector is tailored towards game-playing agents. Consequently, we do not use this reward vector, but the learned features Φ in a goal h. Starting from the first goal h in $\mathcal{H}$, the agent is trained in this environment, and the agents are rewarded according to the features Φ in this goal h. While the agent is interacting with this environment, the agent checks whether the criteria are fulfilled or not. However, the synthetic agent has no genuine experience, and the human-like agent plays on a different level, so the criteria set in the features Φ may be unrealistic. Therefore, we use a criteria threshold c_T to determine the fulfillment condition. Furthermore, when the agent completely fulfills a criterion c of a feature ϕ, we dampen the weights of this feature ϕ. Thus, the agent starts to

perform these interactions less as these are less rewarding. Furthermore, the agent's primary purpose is to fulfill its test goal. Consequently, we reward the agent based on the goal's completion percentage. This extra reward is specified in this thesis as $reward^{completion}$ where the $reward$ is a positive real value and $overallcompletion \in [0, 1]$. To calculate the $overallcompletion$, first, we compute the individual completion of each criterion. Next, we multiply each of these completion scores with each other to calculate the $overallcompletion$. As a result, if the individual completion of a single criterion is zero, the total is also zero. When the agent completes the goal fully or completes a percentage of a goal —given that the number of steps is passed, we decide that the agent achieved the current test goal. The current test goal becomes inactive, and the next test goal becomes active from $\mathcal{H}$. Also, during this goal transition, we reset the current test state.

We propose to use two RL agents, MCTS and Sarsa. In the selection phase of MCTS, we employ transpositions [18] to share information between states. Specifically, we use UCT3 [18]. In the simulation phase of MCTS, we apply knowledge-based evaluations (KBE) to score the states. For the Sarsa agent, we implement the Sarsa(λ) algorithm described in Section 2.1. Eligibility traces help to spread the estimated long-term reward gained by completing a goal. Finally, we selected the Boltzmann exploration strategy [46].

Test Oracle: Defining and performing a test sequence is incomplete if we cannot evaluate whether the test passes or fails. In order to assess if the game performs as anticipated, we propose to use an automated test oracle. Evaluating tests using an automated test oracle is far more efficient since it eliminates the need for human intervention. Therefore, we chose to employ a model-based oracle [95] to assess whether an execution fails or succeeds. We did not employ a vision-based oracle since our goal is not to discover visual faults but to identify differences between game design and implementation.

We initialize the automated test oracle with the constraints —the game model — and the game scenario graph. The test oracle receives the game state and interactions performed at each game step. The oracle first uses the game state and the constraints to assert collision faults. Next, the oracle matches the game state to a node in the game scenario graph. The oracle checks that the transition is correct given the interactions and the game scenario node. Lastly, the oracle checks the game constraints to verify if this transition was made with proper interactions. The oracle reports the found violations if a fault is discovered during any of these steps.

CHAPTER 6

METHODOLOGY III: MCTS MODIFICATIONS

6.1 Transpositions

We propose using transposition tables (TT), specifically the UCT3 approach [18], as our initial adjustment since UCT3 is a proven strategy. In the search tree, we store a reference to an entry in the TT, and then in the TT, we store the node information. This node information is utilized in the selection phase of the MCTS algorithm. Furthermore, this node information is updated during the backpropagation step of the MCTS algorithm. Lastly, the information stored in TT is not used during rollouts of the simulation phase.

6.2 Knowledge-Based Evaluations

Our goal is to test games that may have bugs. Furthermore, these bugs may obstruct reaching terminal states. Testing whether one can lose a game is also a test objective for game tester agents. Consequently, the rewards of penalties received from the game or terminal states could be deceiving. As a result, we propose using KBE to reward our tester agents.

$$\textsc{Eval}_{\text{KBE}}(s,a) = \phi_{(s,a)} \cdot (wd) + (w_h)^{f_{(s,a)}} - (w_h)^{f_s} \tag{3}$$

We use Equation 3 to evaluate the state provided by the environment. The $\phi_{(s,a)}$ represents the features in state s when action a is performed. The weight of achieving a goal state is represented by w_h. Next, $f_s, f_{(s,a)} \in [0,1]$ represents how much of the goal criterion is met. We should add that value of f_s is 0 if f_s is smaller than c_T. Lastly, the dampening factor is represented by d. The dampening factor is used to mitigate the reward obtained from features when the criterion for this feature is already accomplished. KBE enhancement is utilized in all of the tester MCTS agents.

6.3 Tree Reuse

The backpropagation phase of MCTS becomes cumbersome and time-consuming when UCT3 of TT [18] is used. Furthermore, this predicament grows when the tree is reused. Additionally, the test state described in Chapter 4 expands the number of states. For example, attacking the *Walls* does not

progress the game without the test state. However, with the test state, this attacking action is recorded in the test state. As a result, the entire search tree may get relatively large and cannot be reused; thus must be pruned. Pepels et al. [67] employed a rule-based strategy to eliminate the old nodes, and Powley et al. [75] presented a method to recycle old nodes. However, we propose a new approach, namely the *fast expansion* approach. *Fast expansion* is a lightweight tree reuse technique that offers simple interaction with transpositions.

We retain the previously generated tree while generating a new tree with MCTS. We employ *fast expansion* during the selection phase to rapidly gather knowledge from the previous tree. Next, we modify the selection phase of MCTS to search for the selected node in the previous tree. If this node exists in the previous tree, we flatten the information and add this information to the current tree. To flatten a node, first, we calculate the average score $\tilde{X}$, and we set the visitation count of this node to 1. Afterward, we continue with the selection phase, select a new node, and if this node is in the previous tree, we again employ the *fast expansion*. This process is repeated until the selection phase chooses a node absent in the previous tree. As a result, the nodes are expanded in a fast approach, and the previous knowledge is retained. The remaining steps of MCTS remain unaffected. This approach prunes the nodes that are not chosen in the selection phase, thus, preventing bloating of the tree. Most importantly, the previously generated information is not lost.

6.4 MixMax

We propose MixMax because the agents should consider all possible paths, even if these paths are risky. As indicated in Equation 4, Q is used to blend $\tilde{X}_i$, the average score, with the $maxX_i$, maximum score for taking this action, which is given in Equation 2.

$$maxX_i \times Q + \tilde{X}_i \times (1 - Q) \tag{4}$$

The achievement of a test objective is the victory condition for our tester agent. The agent receives the highest reward when the agent reaches this state. Consequently, if the agent has reached this state, MixMax enhancement will increase the likelihood of selecting this action.

6.5 Boltzmann Rollout

We suggest the Boltzmann rollout based on the Boltzmann exploration methodology in RL [89]. However, according to James et al. [42], this modification may generate less effective agents. Therefore, we are interested to see whether this modification will affect the bug-finding stability of our agents.

$$p(i) = \frac{e^{\beta v(s,a)}}{\sum\limits_{a'} e^{\beta v(s,a')}} \tag{5}$$

We modify the rollout phase of the MCTS algorithm by using Equation 5. This equation calculates the probability of selecting a node $p(i)$ in state s. The Boltzmann beta β is the only variable in this equation and controls the stochasticity of selecting a node. When β is set to 0, the Boltzmann rollout

acts like a random rollout. We use the KBE modification, $\text{EVAL}_{\text{KBE}}(s, a)$ Equation 3, to calculate the value $v(s, a)$.

In contrast, James et al. [42] examined why more informed rollouts result in less effective agents. According to their research, knowledge-based rollouts may lead to high bias and low variation, leading to poor performance. However, we decided to implement this modification to see how informed rollouts affect the discovery of bugs.

6.6 SP-MCTS

Single-Player MCTS (SP-MCTS) was developed by Schadd et al. [81]. The UCB1 calculation in Equation 2 is extended by adding the term $\sqrt{\frac{\sum x^2 - n_i \tilde{X}_i + D}{n_i}}$ (see Equation 6). This additional term causes UCB1 to prefer nodes with high variance, and D is usually a big constant value to indicate uncertainty for nodes that are hardly investigated. In puzzle games, Schadd et al. [81] demonstrated that their version outperformed competing techniques like IDA$*$, a variation of A$*$ that restricts the search depth and incrementally increases this depth until the criteria are reached.

$$\tilde{X}_i + C_p \sqrt{\frac{\ln n}{n_i}} + \sqrt{\frac{\sum x^2 - n_i \tilde{X}_i + D}{n_i}} \tag{6}$$

6.7 Computational Budget

One may configure the computational budget of MCTS, which is set as 40ms for the GVG-AI competition. However, a computational budget is not a modification in the traditional sense. Baier and Winands (2016) examined several MCTS time management techniques and studied the effects of the computational budget on five board games. Furthermore, many different computational budgets were evaluated by Nelson [58]. The author discovered that the win rate stabilizes in the GVG-AI framework at a specific computational budget. We find the experimental results on the computational budget are promising. Consequently, propose to investigate the effect of the computational budget on bug-finding behavior.

CHAPTER 7

METHODOLOGY IV: ALTERNATIVE PATH FINDER

RL agents are driven by the feedback they get from their environments. The feedback will shape the agent's policy it receives throughout training. After training, the agent will act according to the learned policy. Additionally, the learned policies will be comparable if we repeatedly train the same agent in the same environment. After each training session, we may assess the trained agent in the same environment to gather trajectories. Because the learned policies were comparable, these trajectories will also be comparable. However, the game designer may be interested in seeing players or agents employing a variety of playstyles.

The agent learns by interacting with the environment. Therefore, the environment's feedback mechanism must be altered to diversify the learned policies. Rewiring the feedback mechanism using a utility function is employed by procedural personas [36, 56]. Consequently, for each persona, the policy that an agent learns will be unique. However, the procedural persona approach falls short when the game designer wants to observe distinct playstyles for the same persona. For instance, if a game has many endings, the game designer could be interested in how various players complete the game. Therefore, she trains an agent that uses the Exit persona's utility weights and then evaluates this agent's execution. Nevertheless, the game's alternative endings will be ignored, and the game creator will only get data corresponding to one possible game ending. One may mask the feedback from some of the alternative endings. However, various paths may lead to the same ending, and this solution fails to address the real problem. Alternatively, the game creator may randomize the agent's behaviors or add random noise to the input, which would help to vary the path. Randomness, however, does not ensure that the agent will create various playtests. To summarize, none of these rudimentary solutions address the problem of how to generate alternative playtests.

On the other hand, the game designer could have asked the human playtesters to alternate their playstyles. Then, the human playtester could use her past knowledge to play the game differently. This playtester knows her path in the previous playtest, and when she is instructed to play differently, she takes a different path. However, the agent is blind to the last executions with RL agents. Therefore, the agents neglect these prior executions. In order to solve this problem, we propose the Alternative Path Finder.

7.1 Measuring Similarity

Markov Decision Process (MDP) could be used to explain a game by formulating the interaction between an actor and the environment [89].

Suppose we obtained the following trajectory from an agent or a human player, $\tau = \{s_0, a_0, s_1, a_1, ..., s_n\}$ where a corresponds to an action, s corresponds to a state, and the subscripts denote the action or state at time t. Current RL agents disregard this previously generated trajectory τ, and our goal is to train an agent that knows the τ. Next, we require this agent to use the τ knowledge to generate a different trajectory than τ. To describe how close these two trajectories are, we must thus compute a metric. To describe how close these two trajectories are, we must thus compute a metric to represent the similarity of these two trajectories. In this study, two different methods are proposed to calculate the similarity. The first method calculates the recoding probability of a state s, $p(s|\tau)$. If $s \in \tau$, the probability should be high, and if $s \notin \tau$, the probability should be low. The second method calculates the prediction error of a dynamics model $q((s_t, a_t, s_{t+1})|\tau)$. If the transition s_t, a_t, s_{t+1} exists in τ, then the prediction error should be low, and if this transition does not exist in τ, then the error should be high.

For the rest of this study, we use observation o in place of state s as the RL agents in most frameworks such as GVG-AI [69] and VizDoom [44] only obtain the observation rather than the state. Furthermore, the observation is the frame f seen by the RL agent playing the game.

7.2 From Recoding Probability to Intrinsic Feedback

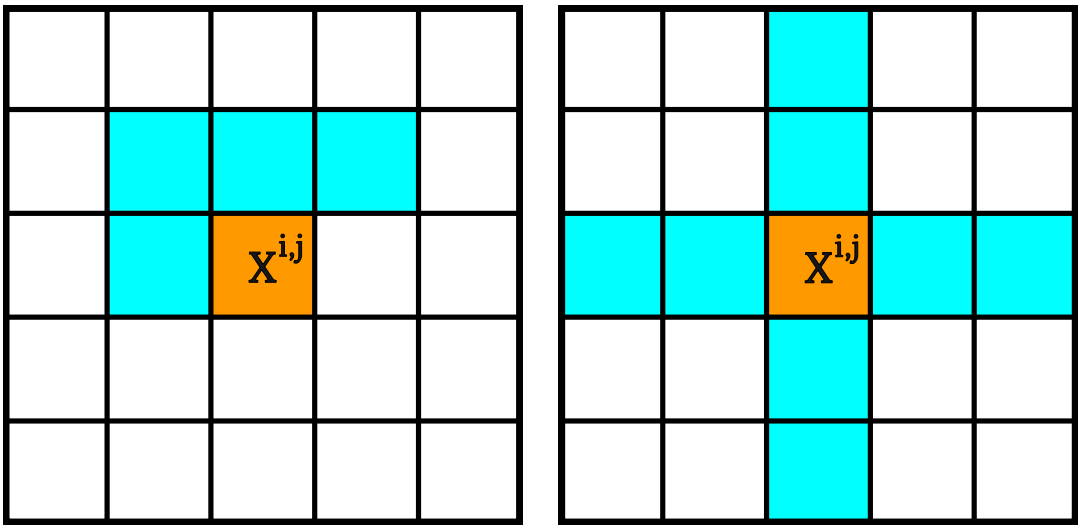
Context Tree Switching (CTS) [12] was utilized by Bellemare et al. [11] to encourage an RL agent for exploration. The recoding probability of a pixel is measured by CTS using a filter. Figure 5a and Figure 5b indicate the filters employed by the CTS and this study, respectively. In order to recode a pixel or to predict, the filter accumulates information about that pixel's neighbors. The recoding probability of an image is computed when this process is performed on each pixel of an image.

We require a boundary probability value to leverage the recoding probability to distinguish between novel and similar frames. This boundary probability is denoted as p_{min} (see Equation 7). We first train a CTS model using all of the frames in previous trajectories. Then we use this trained CTS model to calculate the recoding probability of all the frames in previous trajectories. The lowest of all these recoding probabilities is then assigned as the p_{min}. Since CTS is a learning-positive model, every frame from these trained trajectories will have a greater recoding probability than p_{min}.

$$\begin{aligned} p_{min} = \min(p(f_0|\text{CTS}), p(f_1|\text{CTS}), ..., p(f_n|\text{CTS})) \\ s.t.\ f_{0..n} \in \tau_0, ..., \tau_n \end{aligned} \tag{7}$$

When an agent or a human player plays the game, the actor will receive a new frame f_{new}. We begin by calculating the recoding probability of this frame, p_{new}=$p(f_{new}|\text{CTS})$. If the recoding probability of p_{new} is less than or equal to p_{min}, we conclude that this frame provides new information; otherwise, no new information is provided by this frame. Secondly, we use the difference of p_{new} to p_{min} to compute the amount of reward or penalty.

$$\begin{aligned} p_{new} > p_{min} : feedback = \frac{\beta}{1 + \log \frac{p_{new}}{p_{min}}} - \beta \\ p_{new} \leq p_{min} : feedback = \beta - \frac{\beta}{1 + \log \frac{p_{min}}{p_{new}}} \end{aligned} \tag{8}$$



(a) L-shaped Filter (b) +-shaped Filter

Figure 5: ©2022 IEEE.
Filters mask the pixels around the orange pixel, the data from white pixels are blocked, and the data from the cyan pixels are supplied. Finally, CTS uses the information gathered from cyan pixels to predict the recoding probability of the orange pixel.

To compute the additional reward signal, we utilize Equation 8. When p_{new} is set to 0, this formula produces the highest β reward and the minimum $-\beta$ reward when p_{new} is set to 1. Negative feedback is given for visiting states similar to those already visited, while a positive response is given for visiting novel states. The APF approach that makes use of CTS is referred to as APFCTS.

7.3 From Predicting Dynamics to Intrinsic Feedback

The Intrinsic Curiosity Module (ICM) was used by Pathak et al. [66] to encourage an RL agent for exploration. Neural Network-based architecture ICM learns to anticipate environment dynamics and exploits prediction error as intrinsic motivation. Two neural networks compose ICM: the forward model and the inverse model. Using the current state features $\phi(s_t)$ and current action a_t, the forward model predicts the next state features $\phi(s_{t+1})$. Using the current state features $\phi(s_t)$ and the next state features $\hat{\phi}(s_{t+1})$, the inverse model predicts the current action a_t. ICM uses Convolutional Neural Network (CNN) to encode the states into state features $\phi(s_t) = \text{CNN}(s_{t+1})$. The prediction error is the difference between the extracted future state features $\phi(s_{t+1})$ and the projected next state features $\hat{\phi}(s_{t+1})$. Therefore, the prediction error will be low if the agent has seen the transition $\phi(s_t), a_t, \phi(s_{t+1})$, and large otherwise.

We need a boundary value to utilize the prediction error to distinguish between novel and similar frames. This value is referred to as q_{mean} (see Equation 9). To begin, we construct the ICM architecture. Then, we set the weights of the CNN layers via transfer learning and freeze these layers. As the source for transfer learning, we use the CNN layers of the previously trained agent. Afterward, we train the forward and inverse layers of ICM by using previously collected trajectories as input. When

the training is done, we get an ICM model that predicts the transitions in the collected trajectories with higher certainty and predicts the unseen transitions with a higher fault. Finally, we replay the prior trajectories, collect all prediction errors, and compute the *mean* of all prediction errors.

$$\begin{aligned} q_{mean} = \text{mean}(\text{ICM}(f_0, a_0, f_1), ..., \text{ICM}(f_{n-1}, a_{n-1}, f_n)) \\ s.t.\ f_{0..n} \in \tau_0, ..., \tau_n \\ s.t.\ a_{0..n-1} \in \tau_0, ..., \tau_n \end{aligned} \tag{9}$$

When an actor acts a on frame f, the environment responds to this action and updates its state, and the actor observes f_{new}, a new frame. ICM predicts the error of transition from f to f_{new} as q_{new}=ICM(f, a, f_{new}). If q_{new} is greater than q_{mean}, then the transition of f to f_{new} with action a is unlikely to be in the previous trajectories. On the other hand, if q_{mean} is greater than q_{new}, then the transition of f to f_{new} with action a is likely in the previous trajectories.

$$\begin{aligned} q_{new} > q_{mean} : feedback = \beta - \frac{\beta}{1 + \log \frac{q_{new}}{q_{mean}}} \\ q_{new} \leq q_{mean} : feedback = \frac{\beta}{1 + \log \frac{q_{mean}}{q_{new}}} - \beta \end{aligned} \tag{10}$$

To determine the extra reward signal, we utilize Equation 10. When q_{new} is set to 0 this equation yields $-\beta$ and when q_{new} is set to ∞ this formula yields β. We penalize similar transitions and reward novel transitions using this calculated feedback signal. The APF approach that makes use of ICM is referred to as APFICM.

7.4 APF Architecture

We add a new entity to the Agent and Environment interaction, which is represented in Figure 6. Both APFCTS and APFICM may be used to define the APF. As a requirement, the APF should have been trained using the previous trajectories as described in Section 7.2 or Section 7.3. This requirement is to ensure that APF recognizes states or transitions. When an agent interacts with the environment, APF is the first to receive the new observation and the reward signal from the environment. APF modulates this reward by calculating auxiliary feedback (see Equation 8 or Equation 10).

One of the flaws of this APF architecture is that the feedback is limitless. The agent may cycle through new states or get trapped in a novel state, i.e., the noisy television [15]. The agent may get addicted to this positive feedback and may not perform the actual task in the environment. The second flaw is that, as in Super Mario Bros. [66], certain game elements may be rigid and provide no alternative paths. Consequently, APF will naively penalize this portion of the game.

For each of these problems with APF architecture, we provide a solution. We suggest limiting the overall reward and penalty modulation to address the first issue. This strategy reduces the potential for endless feedback, and it works as follows: if a state is distinct, APF clamps the reward by the positive cap pos_{cap}. APF yields this clamped reward and then deducts the clamped reward from the pos_{cap}. When the positive cap pos_{cap} reaches zero, APF no longer provides a reward for visiting a novel state. Likewise, when the positive cap pos_{cap} reaches zero, APF no longer provides a reward

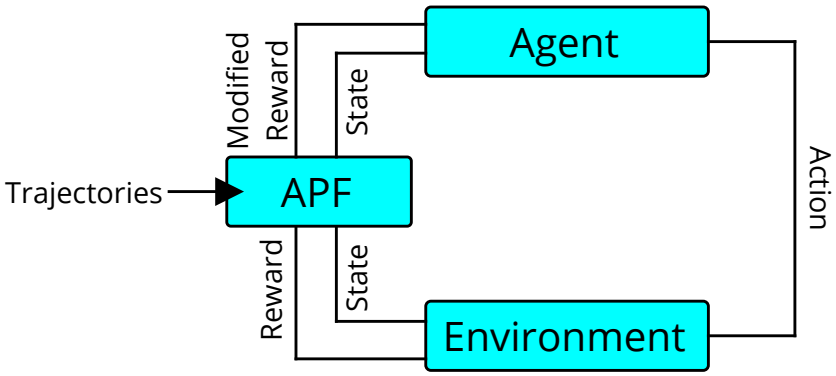


Figure 6: ©2022 IEEE.
Alternative Path Finding Architecture.

for visiting a novel state. We apply the same concepts to the penalty modulation by introducing a negative cap neg_{cap}. By introducing a limit, we mitigate the agent visiting novel states repeatedly [15]. This approach also makes creating the utility function for personas easier since the total reward and penalty are known in advance. Finally, we recommend removing specific segments from the collected trajectories to address the second disadvantage. As a result, APF will not learn to discern states from these segments and will no longer punish the agent.

We presented two distinct APF methods, and each has its merits and disadvantages. For example, the CTS model could be trained on a trajectory of a few frames. Nevertheless, APFICM requires more data than APFCTS. Furthermore, for transfer learning, a previously trained agent must be supplied for APFICM. On the other hand, APFICM is more robust to noise, whereas noise would impact the recoding probability of CTS.

Furthermore, we must establish a line between exploration and APF since APF relies on the techniques utilized in exploration. Exploration approaches aim to expand the agent's environmental awareness while being trained. The exploration methods intrigue the agent to visit unvisited areas of this environment. Consequently, when this agent is evaluated, the agent knows the best actions to perform. On the other hand, APF exploits the exploration methods to learn about the previous trajectories. During training, APF modifies the reward structure of the environment so that the previously played paths are no longer interesting for the agent. When we evaluate the agent, the agent generates a different trajectory than previous trajectories.

CHAPTER 8

METHODOLOGY V: DEVELOPING PERSONA

An archetypal player's decision model is reflected by a persona. For a persona to evaluate a video game, the decision model of the persona should be translated to game variables. Next, this translation should be realized by an actor. Researchers [36, 56] suggested employing a utility function to relate the decision model to game conditions. This utility function substitutes the environment's reward mechanism with a customized reward mechanism for each persona. Researchers have employed RL agents as actors [36, 56]. Therefore, the decision model of a persona is represented by these RL agents, who are comparable to synthetic playtesters. The procedural personas used by these playtesters represented several personas, including the Monster Killer, Treasure Collector, and Exit personas. We enhance the procedural persona approach by introducing a multi-goal persona.

We propose a multi-goal persona to generate a more customizable playtester. We have two reasons why a multi-goal persona would benefit game designers. First, the game designer does not have granular control over the procedural personas. For example, the game designer may want to playtest a procedural monster killer persona that kills monsters until their health drops below a certain percentage. However, when to cease killing monsters was left to the RL agent, and the game designer had little control over these decisions [13]. Second, the procedural personas do not allow development in the decision model. Though procedural personas may realize the persona archetypes that Bartle [9] presented, procedural personas cannot realize the development sequences that Bartle demonstrates. For example, if the goal of the procedural persona is killing monsters, the procedural persona will always be a Monster Killer. This procedural persona will never develop as a Treasure Collector.

A multi-goal persona is a procedural persona having a connected series of objectives rather than a single utility function. A goal consists of a utility function and a transition to the next goal. The transition does not need to be defined if the sequence has one goal. Consequently, a procedural persona is analogous to a goal-based persona with a single goal. The transition links the objectives and occurs when the criteria conditions are fulfilled. Criteria may be, for example, killing 50% of the creatures, exploring 90% of the game, having less than 20% health, or a combination of these factors. The developing persona keeps track of interactions such as how many *Monsters*, and *Treasures* have been slain or gathered. Aside from interactions, the evolving character is aware of how much of its health remains. This information is used by developing personas to determine if the existing criteria are met. When all of the conditions for the current goal are met, the next goal is activated. The training or evaluation of the goal-based persona finishes when no more goals exist.

While presenting the transitions between objectives, we discussed the "sudden" transitions between goals, in which the current goal goes inactive, and the new goal becomes active immediately. This



Figure 7: ©2022 IEEE.
An example level created by GVG-AI framework.

shift, though, can potentially be "fuzzy." Both the current goal and the future goal may be active at the same time. The criterion fulfillment percentage might be used in a fuzzy transition implementation. For example, if the criteria are met by at least 50%, the next goal may become active while the present goal remains active. The persona would benefit from both utility functions. When the persona completes the current objective, the subsequent goal becomes the sole active goal. As a result, a fuzzier transition would result in a smoother progression of playstyles.

We constructed an example level in Figure 7 to showcase the goal-based personas. The *Avatar* may execute the following actions: *Pass*, *Attack*, *Left*, *Right*, *Up*, and *Down*. A pink triangle indicates the *Avatar*'s orientation. If the *Avatar*'s and the action's directions align, the *Avatar* advances one square in that direction; otherwise, the *Avatar* changes direction. When *Avatar* conducts *Attack*, the *Avatar* slashes in the direction of the *Avatar*. *Avatars* can attack *Monsters* and kill them. The *Monsters* move randomly and kill the *Avatar* if they collide. *Avatar* may also pick up Treasure chests by simply moving over them. Finally, the game ends successfully when the *Avatar* escapes through the *Door*.

In order to playtest a Monster Killer persona, a game designer may conceptualize the following personas. The objective of the first Monster Killer persona is to kill the *Monsters* and then collect the *Treasures* as a prize. The second Monster Killer gathers the *Treasure* to gain an edge over the *Monsters* and then kills the *Monsters*. The game designer creates two developing personas to implement the aforementioned personas, as seen in Figure 8. She then creates the utility functions for the objectives, as seen in Table 1. The game designer may use any RL agent to create these personas as playtesters. The game designer can employ the agent for playtesting when the agent has completed training. Developing personas is important because it provides a mechanism to define how players' objectives alter throughout a game.

Table 1: ©2022 IEEE.
Utility weights for the goals.

	Goal Names		
Game Event	Killer	Collector	Exit
Death	-1.0	-1.0	-1.0
Exit Door			1.0
Monster Killed	1.0		
Treasure Collected		1.0	

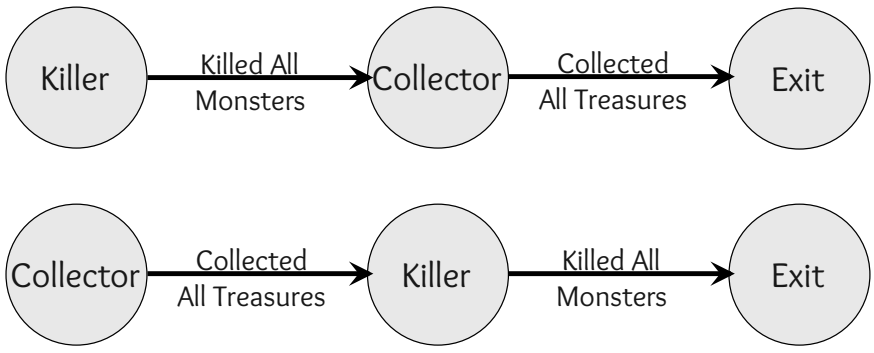


Figure 8: ©2022 IEEE.
Developing Persona.

CHAPTER 9

EXPERIMENTS

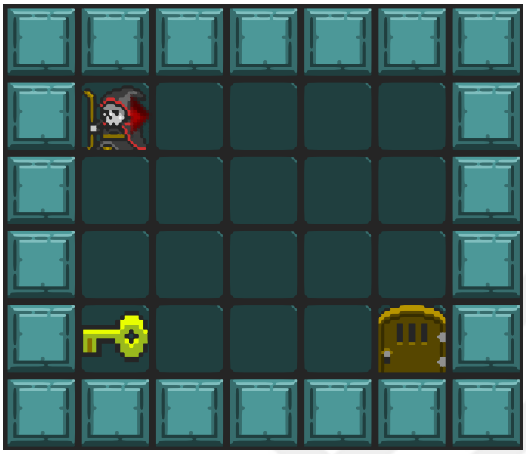
9.1 Experiments for Synthetic and Human-like Test Goals

To experiment with synthetic and human-like test goals, we created three distinct games using the GVG-AI framework (see Figure 9, Figure 10, and Figure 11). These games range in difficulty and have dimensions of 6x7, 8x9, and 10x11. These games are referred to as Game A, Game B, and Game C, respectively. There are four levels in each of these games. The arrangement of these stages varies, which are shown in Figure 9, Figure 10, and Figure 11. In addition, the *Avatar* may only utilize the *Use* action in the first two levels of each game. The objective of Game A is straightforward: the *Avatar* must pick up the *Key* and proceed to the *Door*. The *Avatar* in Game B must extinguish the *Fire*, pick up the *Key*, and complete the level by passing through the *Door*. The *Avatar* in Game C must construct the *Key* by assembling the *Key* pieces, the mush pick up the assembled *Key*, and complete the level by passing through the *Door*. Finally, each level of Game B includes a unique game scenario graph.

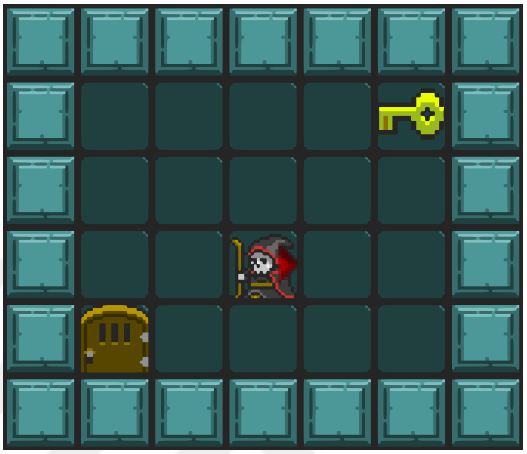
For our experiments, we made minor modifications to the GVG-AI code to access all of the interactions amongst all sprites, including the hidden sprites. Although hidden sprites are not necessary for gameplay, this information might be crucial while testing a game and could help the agent discover bugs. Furthermore, the percentage of a level explored and how it is explored defines different tester behavior. For example, a tester might explore all of the tiles of a game without interacting with any sprite other than $Floor$.

To validate our technique, we employ fault seeding. According to Ammann and Offutt [3], fault seeding is a technique for evaluating the defect detection rate of software tests and test processes. The VGDL descriptions (see Figure 12, Figure 13, and Figure 14) represent the source code of our games—not the GVG-AI engine; hence, we modified VGDL descriptions while injecting errors. These faults cause the game's implementation to act differently than the game's ideal concept. To diversify the bugs, we utilize [49] as a reference. We change the VGDL game description by eliminating lines from the interaction set, rearranging or renaming the sprites in the interaction, and adding fallacious interactions. We introduced 45 defects into the VGDL scripts of the games. If there are many occurrences of the same problem during scoring, it is scored as one. The fault injected VGLD descriptions are shown in Figure 12, Figure 13, and Figure 14.

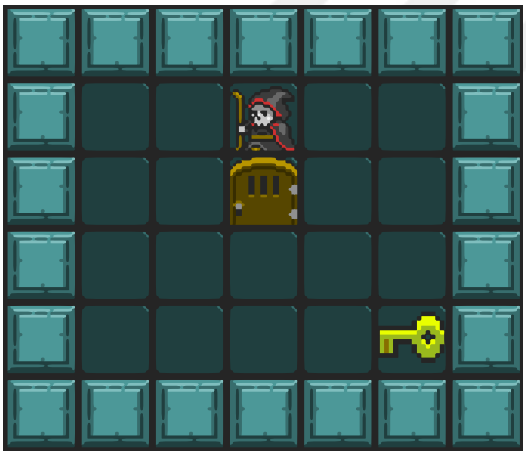
A total of 427 trajectories from 15 human volunteers with various gaming and testing backgrounds were gathered. During data collection, we specified the game rules and goals for each game, but there were no directions on what to test. Testers could replay the same level as often as they wanted and switch between levels and games. There were tutorial levels to help players become acquainted with



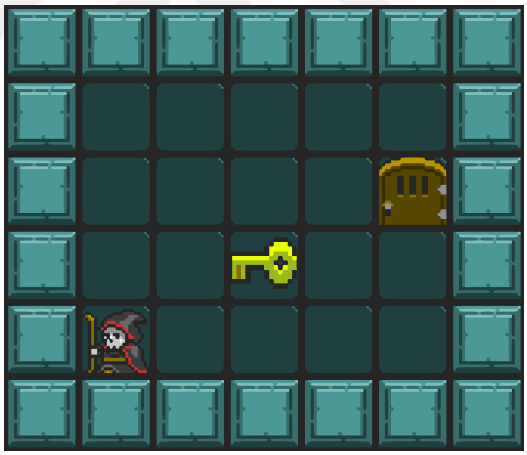
(a) Game A Level 1



(b) Game A Level 2

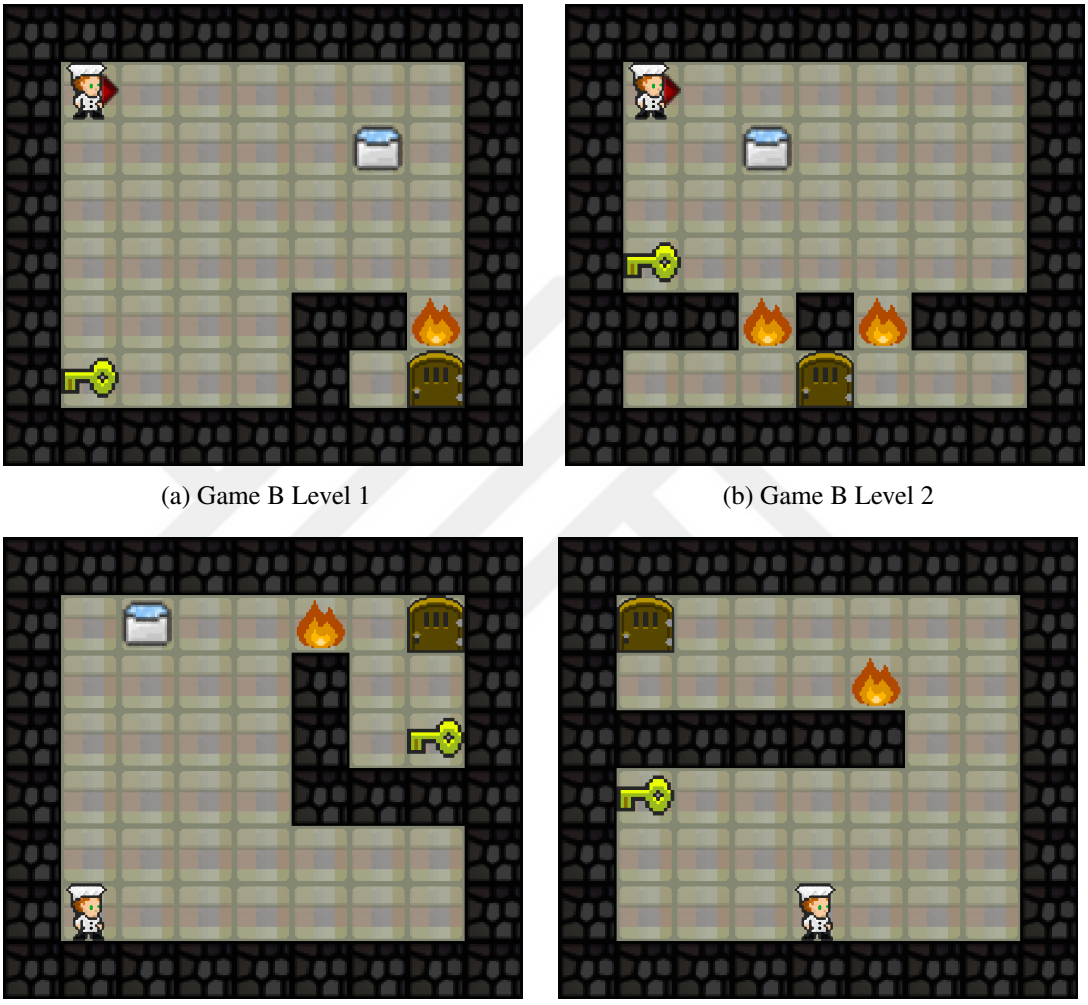


(c) Game A Level 3



(d) Game A Level 4

Figure 9: ©2019 IEEE.
Four levels of Game A.



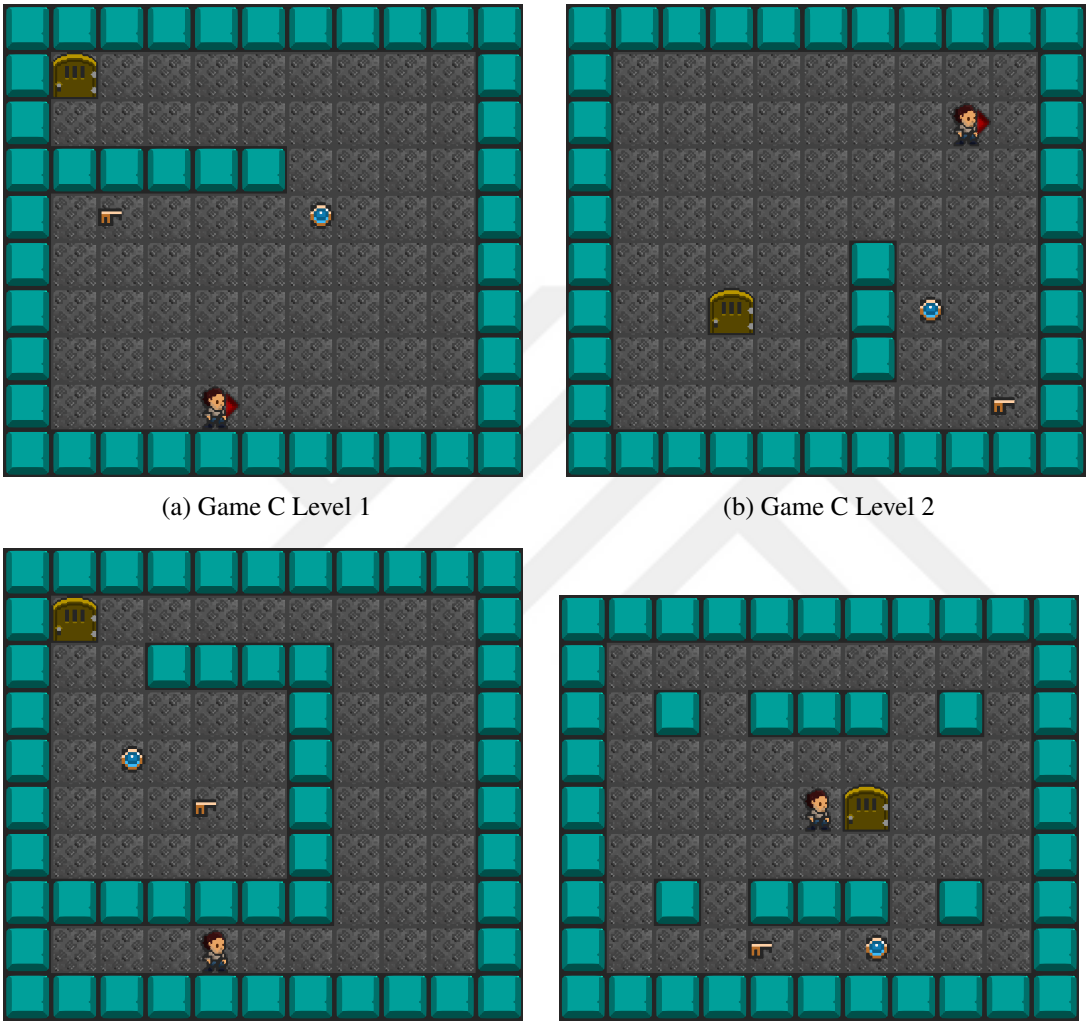
(a) Game B Level 1

(b) Game B Level 2

(c) Game B Level 3

(d) Game B Level 4

Figure 10: ©2019 IEEE.
Four levels of Game B.



(a) Game C Level 1

(b) Game C Level 2

(c) Game C Level 3

(d) Game C (9x11) Level 4

Figure 11: ©2019 IEEE.
Four levels of Game C.

```
BasicGame square_size=60
SpriteSet
  floor > Immovable img=oryx/floor3
  goal >
    goal2  > Door color=GREEN img=oryx/doorclosed1
    goal1  > Door color=GREEN img=oryx/doorclosed1
  key   > Immovable color=ORANGE img=oryx/key2
  sword >
    swordnokey > OrientedFlicker limit=9 singleton=True img=oryx/slash1
    swordkey > OrientedFlicker limit=9 singleton=True img=oryx/slash1
  avatar  > ShootAvatar
    nokey   > img=oryx/necromancer1 stype=swordnokey
    withkey > color=ORANGE img=oryx/necromancerkey1 stype=swordkey
  wall > Immovable autotiling=false img=oryx/wall3

LevelMapping
  g > floor goal2
  + > floor key
  A > floor nokey
  w > floor wall
  . > floor

InteractionSet
  avatar wall  > stepBack
  nokey goal2    > stepBack
  nokey goal1    > stepBack
  wall swordnokey > killSprite scoreChange=0
  goal1 swordkey > killSprite scoreChange=0
  goal2 swordkey > spawn stype=goal1 scoreChange=0
  goal2 swordkey > killBoth scoreChange=0
  goal2 withkey  > killSprite scoreChange=0
  nokey key     > transformTo stype=withkey scoreChange=0 killSecond=True

TerminationSet
  SpriteCounter stype=goal   win=True
  SpriteCounter stype=avatar win=False
```

Figure 12: ©2019 IEEE.
VGDL describing the ruleset for Level 1 of Game A.

```
BasicGame square_size=60
SpriteSet
  floor > Immovable img=newset/floor6
  goal >
    goal2  > Door color=GREEN img=oryx/doorclosed1
    goal1  > Door color=GREEN img=oryx/doorclosed1
  key   > Immovable color=ORANGE img=oryx/key2
  sword >
    swordnokey > OrientedFlicker limit=9 singleton=True img=oryx/slash1
    swordkey > OrientedFlicker limit=9 singleton=True img=oryx/slash1
  debris > Immovable autotiling=false img=newset/whirlpool1
  avatar  > ShootAvatar
    nokey   > img=newset/chef stype=swordnokey
    withkey > color=ORANGE img=newset/chef_key stype=swordkey
  wall > Immovable autotiling=false img=newset/floor4
  water > Immovable autotiling=false img=oryx/water1
  fire > Passive autotiling=false img=oryx/fire1

LevelMapping
  g > floor goal1
  + > floor key
  e > floor water
  f > floor fire
  A > floor nokey
  w > floor wall
  . > floor


InteractionSet
  withkey water > killIfFromAboveNotMoving
  water avatar > bounceForward
  water wall goal key > undoAll

  avatar fire > killSprite

  fire water > transformTo stype=debris scoreChange=0 killSecond=True
  avatar wall  > stepBack
  nokey goal1    > stepBack

  goal1 withkey  > killSprite scoreChange=0
  nokey key     > transformTo stype=withkey scoreChange=0 killSecond=True
  water swordnokey > transformTo stype=fire  killSecond=True


TerminationSet
  SpriteCounter stype=goal   win=True
  SpriteCounter stype=avatar win=False
```

Figure 13: ©2019 IEEE.
VGDL describing the ruleset for Level 1 of Game B.

```
BasicGame square_size=60
SpriteSet
  floor > Immovable img=newset/floor2
  goal > Door color=GREEN img=oryx/doorclosed1
  key   > Immovable color=ORANGE img=oryx/key3
  keyleft   > Immovable color=ORANGE img=oryx/key3_0
  keyright   > Immovable color=ORANGE img=oryx/key3_1
  sword > OrientedFlicker limit=9 singleton=True img=oryx/slash1
  avatar  > ShootAvatar
    nokey   > img=newset/man2 stype=sword
    withkey > color=ORANGE img=newset/man2_key stype=sword
  wall >
    normalwall > Immovable autotiling=false img=newset/blockT
    fakewall > Immovable autotiling=false img=newset/blockT

LevelMapping
  g > floor goal
  r > floor keyright
  l > floor keyleft
  A > floor nokey
  w > floor normalwall
  t > floor fakewall
  . > floor

InteractionSet
  keyleft avatar > bounceForward
  keyright avatar > bounceForward
  keyleft goal normalwall > undoAll
  keyright goal normalwall fakewall > undoAll

  keyleft keyright > transformTo stype=key killSecond=true
  avatar wall > stepBack

  key sword > killSprite
  nokey goal > stepBack
  goal withkey  > killSprite scoreChange=0
  nokey key     > transformTo stype=withkey scoreChange=0 killSecond=true

TerminationSet
  SpriteCounter stype=goal   win=True
  SpriteCounter stype=avatar win=False
```

Figure 14: ©2019 IEEE.
VGDL describing the ruleset for Level 1 of Game C.

the controls and the surroundings. We should point out that all games included bugs except for the instructional levels. We gathered trajectories using the GVG-AI framework, and for each game, a total of 118, 173, and 136 trajectories were collected, respectively. It should be mentioned that 8x9 has more game scenario pathways than the other games. Hence testers performed more tests on it.

For these trajectories, we used MGP-IRL with κ_T likelihood thresholds of 0.0, 0.5, and 1.0; note that MLIRL calculates log-likelihood. We used the following parameters for MLIRL described in [7, Algorithm 1]: M=20, β=5, T=0.01. To evaluate the efficiency of the MGP-IRL, we evaluated these three alternative threshold values. We suggest using a threshold of 0.0 to discover weights, and a threshold of 1.0 is nearly as effective as using MLIRL. For each game, for each tester, and each level, a human-like test goal is learned using the collected trajectories from the other three levels. As a result, human-like agents produce more tests than the original human tester.

We used the game scenario graph and the sprites to create the synthetic test goals. The synthetic agent generated 28, 234, and 88 distinct test sequences for our Game A, Game B, and Game C. We used entire path coverage because there are no loops in these game scenario graphs. Finally, a baseline agent is developed to assess the effect of modifications on bug detection. The test goals used by the baseline agent are generated using only the game scenario graph.

Our comparison did not employ off-the-shelf testing methods such as record/replay, test automation frameworks, or monkey testing. Even if the avatar faces a different way, the record/replay tools will not work. A specialist must manually create the scenarios for test automation frameworks, which are laborious and not scalable. Finally, monkey testing creates random events to stress test the user interface, which is not the goal of our test. As a result, COTS testing methods were insufficient in our studies.

For Sarsa(λ), we chose the following parameters: learning rate α=0.03, γ=0.95, β=1, temporal difference λ=0.8, and for MCTS: γ=0.95, exploration term C_p=3, rollout depth is 8, 300ms for computation budget on i7-4700HQ using single core. Five MCTS runs were performed. Our criteria threshold c_T was set to 0.01, the goal completion reward was set to 10, and the features other than non-zero reward are considered as a singular feature with a reward of -1. We have decided not to advance an agent if it does not achieve its present goal. We evaluated with game durations of 50, 100, 150, 200, 250, and 300 because there is no clear definition of a terminal state and picked the one with the highest criteria fulfillment. We set the direction option for Game C to *All* to reduce the memory requirements. Sarsa(λ) agent ran for a few minutes to 6 hours, depending on the intricacy of the goal sequence being played. Finally, several bugs enabled testers to leave the designated grid area. We assumed that the agent was interacting with the *Floor* sprite since the outside of the grid was not modeled. This behavior generated issues such as divergence during training using MCTS and Sarsa agents. As a result, we issued a negative reward when the agent attempted to explore outdoors after exiting the map.

9.2 Experiments for MCTS Modifications

We utilized the same experimentation setup in Section 9.1. To experiment with synthetic and human-like test goals, we created three distinct games using the GVG-AI framework (see Figure 9, Figure 10, and Figure 11). These games range in difficulty and have dimensions of 6x7, 8x9, and 10x11. These games are referred to as Game A, Game B, and Game C, respectively. There are four levels in each

of these games. The arrangement of these stages varies, which are shown in Figure 9, Figure 10, and Figure 11. In addition, the *Avatar* may only utilize the *Use* action in the first two levels of each game. The objective of Game A is straightforward: the *Avatar* must pick up the *Key* and proceed to the *Door*. The *Avatar* in Game B must extinguish the *Fire*, pick up the *Key*, and complete the level by passing through the *Door*. The *Avatar* in Game C must construct the *Key* by assembling the *Key* pieces; then picks up the assembled *Key*, and complete the level by passing through the *Door*. Finally, each level of Game B includes a unique game scenario graph.

From 15 distinct human subjects with a range of gaming and testing backgrounds, we gathered a total of 427 trajectories. The testers warmed up by playing practice levels to become acquainted with the game controls and environment. During testing, participants could play the games in any sequence and any number of times. The GVG-AI framework is used to collect the tester trajectories.

These gathered trajectories are used to generate human-like test goals. Human-like agents employed the human-like test goals collected from the other three levels of the same game during tests. We developed synthetic test goals by generating paths from a level's game graph. The sprite set of the game under test is used to build a modification list. This modification list is used to make changes to the sampled pathways. Finally, the original test goals are utilized as baseline test goals.

We created five different MCTS agents using the modifications described in Section 6 and ran them five times for each level. These agents are KBE-MCTS, FE-MCTS, MM-MCTS, BR-MCTS, and SP-MCTS (for modifications, see Table 2). The MCTS agents all utilized γ=0.95 and a rollout depth of 6. In all MCTS agents except SP-MCTS, the exploration term is set to C_p=3.0. We evaluated with 40 and 300 milliseconds for the computational budget on an i7-8750H (4.1 GHz) with a single core. An automated test oracle examines each test sequence after being created by these agents.

We use the following parameters for KBE enhancement to calculate Equation 3 threshold for the criteria c_T=0.01, goal reward w_h=10, and for interacting with the features that are absent in goal $w = -1$. For the MixMax enhancement, Q is chosen as 0.25. Furthermore, in order to retain some randomness in Boltzmann rollouts, we selected Boltzmann beta as $\beta = 0.5$. Lastly, for SP-MCTS, we choose D as 10000.

9.3 Experiments for Developing Persona and APF

To test our ideas, we utilized the environments GVG-AI [69], and VizDoom [44]. This section describes the environments and the experimental setup.

Figure 15 shows the first testbed game produced using the GVG-AI framework. The game has a 14×20 grid size and includes an *Avatar*, *Exits*, static *Monsters*, *Treasures*, and *Walls*. A human player or an agent controls the *Avatar*. The game continues until the *Avatar* goes to one of the *Exits*, or is killed by a *Monster*, or until 200 timesteps have passed. The action space has six actions: *No-Op*, *Attack*, *Left*, *Right*, *Up*, and *Down*. The GVG-AI framework is extended to handle a game with several *Doors*. The following interactions result in separate feedback for the actor: killing a *Monster*, being killed by a *Monster*, collecting a *Treasure*, and colliding with a *Door*.

The second testbed game is a Doom level, as illustrated in Figure 17. The game has a 1600×832 grid size and includes an *Avatar*, *Exit*, *Monsters*, *Treasures*, and *Walls*. A human player or an agent

Table 2: ©2020 IEEE.
The MCTS Agents' Modifications.

	Agents				
	KBE - MCTS	FE - MCTS	MM - MCTS	BR - MCTS	SP - MCTS
Modifications					
Transpositions	✓	✓	✓	✓	✓
KBE	✓	✓	✓	✓	✓
Tree Reuse		✓			
MixMax			✓		
Boltzmann Rollout				✓	
SP-MCTS					✓

controls the *Avatar*. The game continues until the *Avatar* goes to the *Door*, is slain by a *Monster*, or reaches 2000 timesteps. *Attack*, *Move Left*, *Move Right*, *Move Up*, *Move Down*, *Turn Left*, and *Turn Right* are the seven actions in the action space. The following events result in separate feedback for the actor: killing a *Monster*, being killed by a *Monster*, collecting a *Treasure*, and colliding with the *Door*. Furthermore, the actor receives negative feedback of 0.001 for every step performed.

The third testbed game is another Doom level, as depicted in Figure 18. The game has a 1664×704 grid size and includes an *Avatar*, *Exit*, and *Walls*. A human player or an agent controls the *Avatar*. The game continues until the *Avatar* walks through the *Door*, or until 2000 timesteps have passed. *Attack*, *Move Left*, *Move Right*, *Move Up*, *Move Down*, *Turn Left*, and *Turn Right* are the seven actions in the action space. If the actor interacts with the *Door*, he or she receives feedback. Furthermore, the actor constantly receives negative feedback of 0.001 for every step performed.

We experiment with procedural and goal-based personas in the first and second testbed games. In the first and third testbed games, we evaluated the APF. For the APF experiment, we utilized the same random seed to ensure that the APF approach was correctly tested. In all experiments, we utilize the PPO [83] agent. We modified the base PPO implementation for the PPO+CTS, PPO+ICM, PPO+APFCTS, and PPO+ICM+APFICM. We used the Stable-Baselines project [35] as our base PPO agent. During our initial experiments, we tested the proposed approaches with other RL agents, and we found that PPO requires less hyperparameter tuning. Hence, we used PPO in all of our experiments. Table 3 shows the hyperparameters of PPO agents, whereas Table 4 shows the hyperparameters of APF approaches. Lastly, since the initial game is deterministic, we assessed the trained agent once. Nevertheless, because the second and third games are stochastic, we tested the trained agent 1000 times. Additionally, we found that our training was more reliable if an exploration algorithm like CTS or ICM was utilized. The training in the first game had to be restarted for the agents that do not use exploration.

The GVG-AI environment sends observations with the shape $160 \times 112 \times 4$ is sent by the GVG-AI environment. We scale them down to 80×56 and then turn them into grayscale. Following that, we

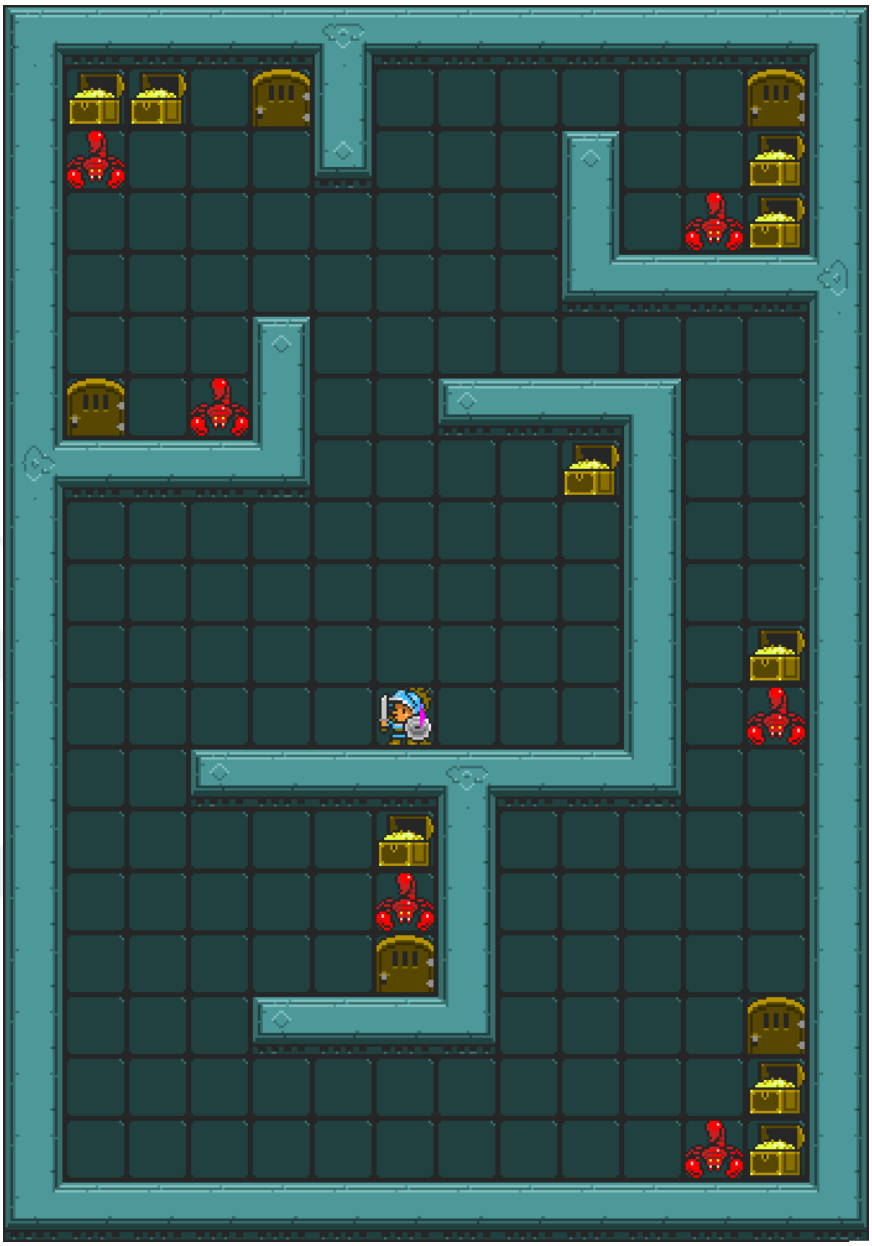


Figure 15: ©2022 IEEE.
Map of the first testbed game.

stack the most recent four observations before feeding the stacked data to the agent. Next, we transform the observation into a 42×42, 3-bit grayscale picture for CTS utilized in PPO+CTS and APFCTS and then compute the recoding probability of this observation. Finally, we resize the observation sent by the Doom environment, which has the following dimensions: $160 \times 120 \times 1$ to $84 \times 84 \times 1$, and feed the downsized observation to the agent and APFICM.

We created four distinct procedural personas and five distinct developing personas. The four procedural personas are Exit, Monster Killer, Treasure Collector, and Completionist. Table 5 shows the utility



Figure 16: ©2022 IEEE.
Doom in-game snapshot.

weights of these procedural personas. These procedural personas were selected from [36], and we used these personas as models to create their developing persona equivalents. The five developing personas are Developing Monster Killer, Developing Treasure Collector, Developing Raider, Developing Completionist, and Developing Casual Completionist. Table 7 shows the development sequences of these personas, Table 6 shows the utility function of the goals, and Table 8 shows the criteria of these goals.

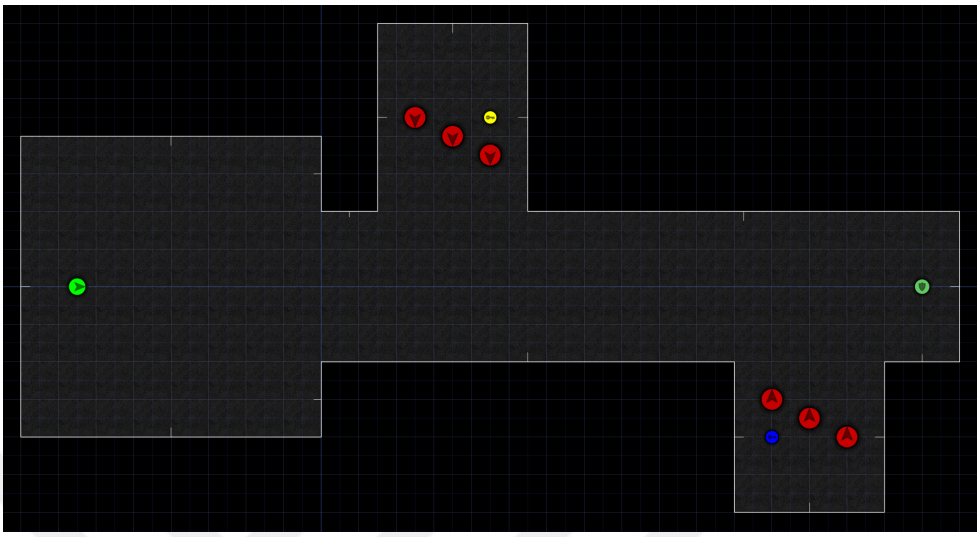


Figure 17: ©2022 IEEE.
Map of second testbed game.

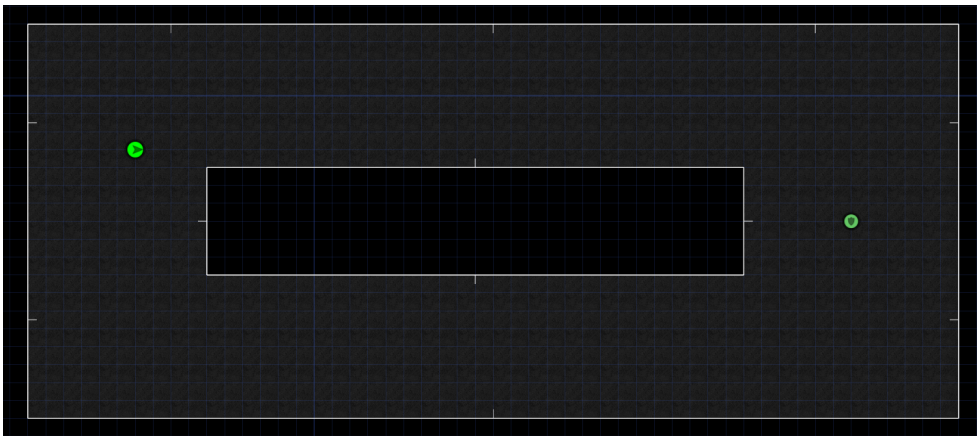


Figure 18: ©2022 IEEE.
Map of third testbed game.

Table 3: ©2022 IEEE.
Hyperparameters of RL Agents.

	Agents		
Hyperparameters	PPO	PPO+CTS	PPO+ICM
Policy	CNN	CNN	CNNLstm
Timesteps	1e8	1e8	2e8
Horizon	256	256	64
Num. Minibatch	8	8	8
GAE (λ)	0.95	0.95	0.99
Discount (γ)	0.99	0.99	0.999
Learning Rate (α)	5×10^{-4}	5×10^{-4}	5×10^{-4}
Num. Epochs	3	3	4
Entropy Coeff.	0.01	0.01	0.001
VF Coeff.	0.5	0.5	0.5
Clipping Param.	0.2	0.2	0.1
Max Grad. Norm.	0.5	0.5	0.5
Num. of Actors	16	16	32
CTS Beta (β)	-	0.05	-
CTS Filter	-	L-shaped	-
ICM State Features	-	-	256
ICM Beta (β)	-	-	0.2

Table 4: ©2022 IEEE.
Hyperparameters of APF Methods.

Hyperparameters	APFCTS	APFICM
pos_{cap}	0.4	0.1
neg_{cap}	-0.4	-0.4
APF Beta (β)	0.01	0.01

Table 5: ©2022 IEEE.
Utility weights of game events for the Exit (E), Monster Killer (MK), Treasure Collecter (TC), and Completionist (C) procedural personas.

	Personas			
Game Event	(E)	(MK)	(TC)	(C)
Reaching an Exit	1	0.5	0.5	
Killing a Monster		1		1
Collecting a Treasure			1	1
Dying	-1	-1	-1	-1

Table 6: ©2022 IEEE.
Utility weights of game events for Killer (K), Collecter (Col), Exit (E), and Completionist (Com) goals.

	Goal Names			
Game Event	(K)	(Col)	(E)	(Com)
Death	-1	-1	-1	-1
Exit Door			1	
Monster Killed	1			1
Treasure Collected		1		1

Table 7: ©2022 IEEE.
Development sequences for developing personas.

Hyperparameters	**Development Sequence**
Dev. Killer	Killer -> Exit
Dev. Collector	Collector -> Exit
Dev. Raider	Killer -> Collector -> Exit
Dev. Completionist	Completionist -> Exit
Dev. Casual Completionist	Casual Completionist -> Exit

Table 8: ©2022 IEEE.
Goal criteria for Killer (K), Collecter (Col), Completionist (Com), and Casual Completionist (Cas. Com.) personas.

	Goal Names			
Criterion	(K)	(Col)	(Com)	(Cas. Com.)
Monsters Killed	50%		100%	
Treasure Collected		50%	100%	
Remaining Health				50%

CHAPTER 10

RESULTS

10.1 Results for Synthetic and Human-like Test Goals

We evaluated eight distinct tester groups to evaluate bug finding performance: original human testers, three human-like Sarsa(λ) agents, one MCTS agent with a likelihood threshold of 0.0, one baseline Sarsa(λ) agent, and one MCTS and one Sarsa(λ) agent with synthetic goals. For bug detecting performance, there are two figures. The first one compares these groups in each game using a barplot. Individual agents contribute to a group's overall score, and the total bug count is considered total unique bugs. The percentage of bugs found by MCTS agents is the mean of 5 runs. The second figure compares individual testers in human-like agents to original human testers. This figure is represented by a violin plot.

The cross-entropy of human behavior and agent behavior is used to assess how similar human-like agents are to original humans. The trajectory collected from the human tester is replayed to get a list of interactions $\zeta^{h}_{0,\ldots,n}$. Then the trajectory executed by a human-like agent learned from this human tester's trajectory is replayed to find a second list of interactions $\zeta^{a}_{0,\ldots,n}$. Each list is binned using $\eta_0, \eta_1, Type, Avatar_{State}$, and the frequency of each bin is used to calculate the cross-entropy of $\zeta^{h}_{0,\ldots,n}$ and $\zeta^{a}_{0,\ldots,n}$. We deleted interactions' position and direction during the comparison because they heavily rely on the level layout. Finally, the number of actions done by the agents and splits performed by MGP-IRL are investigated. Violin plots are used to depict action-length figures. Figures for cross-entropy and the number of splits are displayed in box plots with IQR=1.5, except for cross-entropy, which is shown in the log scale.

Figure 19 bugs found depicts the bug-finding abilities of several agents in each game. We can see that humans discovered all of the bugs in the second game and 90% of the bugs in the first and third games. In Game A, human-like agents with a probability threshold of 0.0 and the synthetic Sarsa(λ) agent outperformed humans and detected all bugs, while the synthetic MCTS agent also found 90% of all bugs. The baseline agent's bugs can be read as the number of bugs discovered by playing only the scenarios supplied by the designer. The total performance difference between the MCTS agent and Sarsa(λ) ranges from 5% to 10%. The contrast between various testers is obvious in Game B. Human-like agents outperformed synthetic agents, and there is a 10% gap between the top human-like agent and human testers. The baseline agent outperforms by more than 40%. In Game C, the gap between humans and agents widens, and the synthetic Sarsa(λ) agent outperforms the human-like Sarsa(λ) agent with a probability threshold of 0.0. Also, fewer defects are discovered in all three games when MCTS agents use their test objectives.

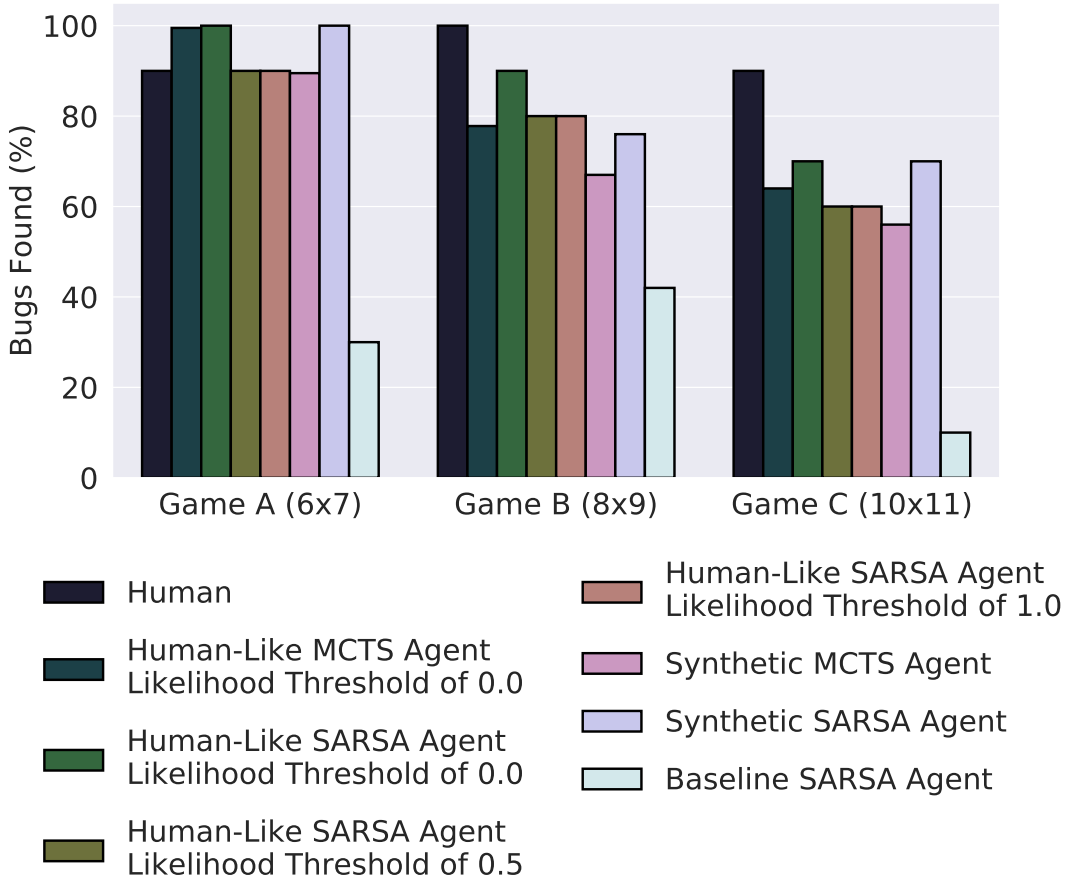


Figure 19: ©2019 IEEE.
The Percentage of Bugs Found by RL Agents and Human Testers.

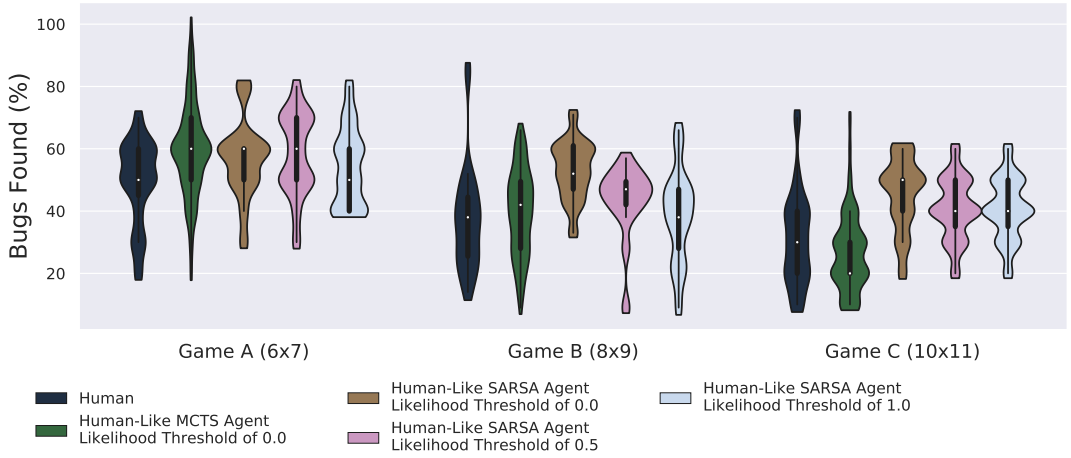


Figure 20: ©2019 IEEE.
Percentage of Bugs Found by Human-Like RL Agents and Human Testers.

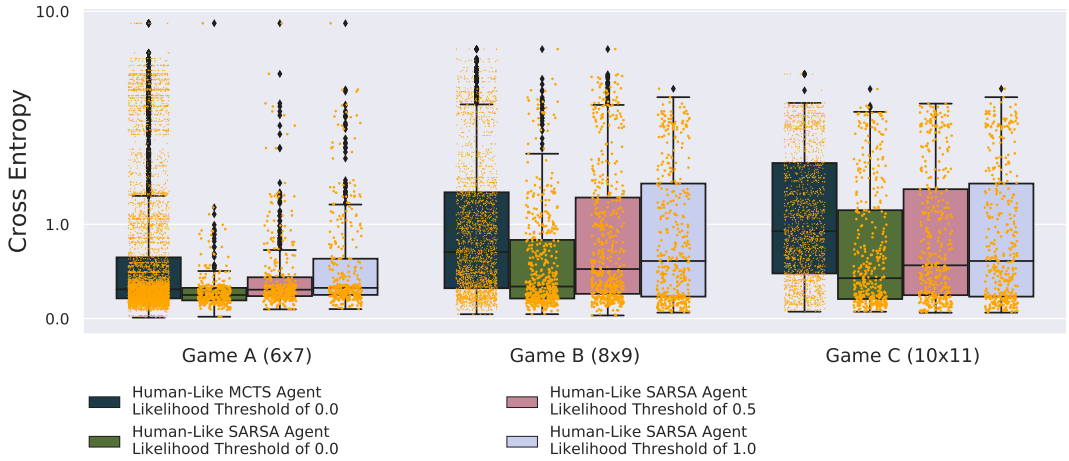


Figure 21: ©2019 IEEE.
Cross-Entropy of Human-Like RL Agents.

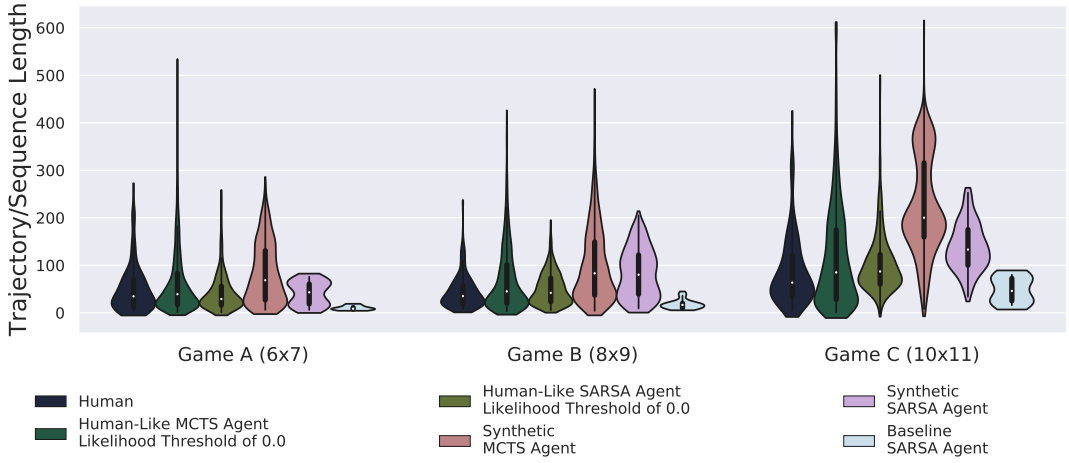


Figure 22: ©2019 IEEE.
Length of Trajectory/Sequence The RL Agents and Human Testers.

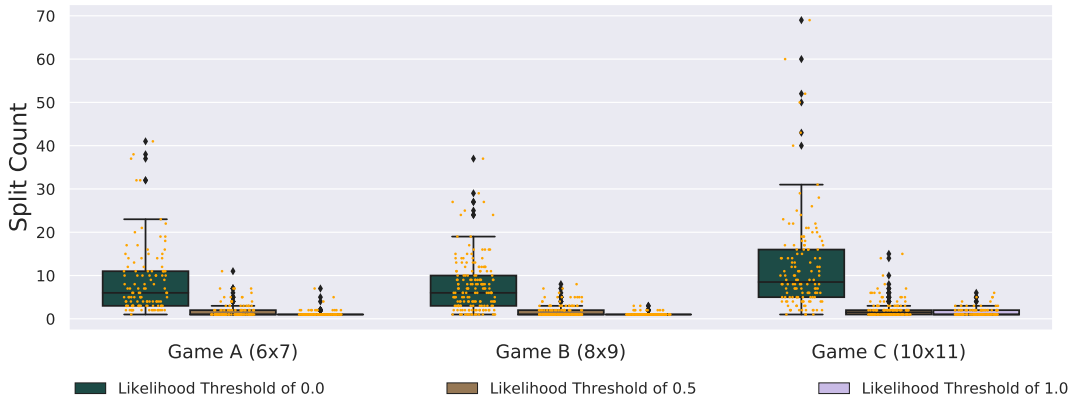


Figure 23: ©2019 IEEE.
Number of Splits Made by MGP-IRL to Collected Human Trajectories.

10.1.1 Experiment 1: Agents Testing Game A

The scores of several human-like entities in detecting bugs are depicted in Figure 20 bugs found (Game A). Individual performance of human-like agents is at least equal to or better than individual humans in a basic game like this, where an agent may achieve almost flawless runs. The human-like Sarsa(λ) agent with a probability threshold of 0.0 conducted the most similar interactions, as seen in Figure 21 (Game A). Additionally, this similarity was negatively impacted by the likelihood threshold. The human-like MCTS agent with a probability threshold of 0.0 executed the most actions among all agents in Figure 22 (Game A). The baseline agent completed the game in fewer than 15 acts. The number of splits produced by MGP-IRL is shown in Figure 23 (Game A): the likelihood threshold of 0.0 splits more than the others, the likelihood threshold of 1.0 splits the least, and most of the time, it considers the trajectory as a whole. Finally, the overall number of bugs discovered —in all five runs— of human-like MCTS, and synthetic MCTS is 100%.

10.1.2 Experiment 2: Agents Testing Game B

Human-like agents have a higher bug-detecting mean than individual human testers in Figure 20 (Game B), but they cannot compete with the best human tester. The cross-entropy of interactions is comparable across the agents in this game, with the human-like agent with a probability threshold of 0.0 leading, as seen in Fig (Game B). Retargeting was not optimal in this game since one of the sprites was missing in the last level. Thus, the total cross-entropy is greater than in Game A. In Figure 22 (Game B), we can observe that the synthetic MCTS agent performed the most actions and had a different shape from the other agents. The maximum number of actions that the baseline agent carried out was 40. According to Figure 23, the human-like agent with a probability threshold of 1.0 executed the fewest splits, while the human-like agent with a the likelihood threshold of 0.0 divided the trajectory the most. Finally, the overall number of flaws detected in all five runs of human-like MCTS is 90%, whereas synthetic MCTS is 76%.

10.1.3 Experiment 3: Agents Testing Game C

Human-like Sarsa(λ) agents increased the overall performance of individual humans, as seen in Figure 20 (Game C), except for one tester. Compared to all testers, human-like MCTS agents have lower mean bug finding performance. Figure 21 (Game C) indicates that the mean cross-entropy of Sarsa(λ) agents is less than 0.5 and similar to previous games. We can see in Figure 22 (Game C) that all actors performed more actions than in previous games. When compared to other games, Figure 23 (Game C) shows that the number of splits for each agent increased. Finally, the total bugs detected —in all five runs— of human-like MCTS are 90%, and synthetic MCTS are 80%, which indicates MCTS could find a different bug in every run.

10.2 Results for MCTS Modifications

We present our results in Table 9 and Table 10. The results of the MCTS agents are displayed within the 0.95 confidence interval since they are run five times. The score of Sarsa(λ) is not inside an interval because it is only run once. When calculating the number of bugs discovered, multiple instances of the same bug are recorded as one. As more than one tester tested a game, *Combined* indicates all of the bugs found by these testers when their results are merged, and *Individual* implies bugs found by each tester. We also prioritize tester agents that uncover the most faults with the quickest test sequences or within the smallest computing budget. Finally, we employed cross-entropy to compare the interactions performed by a human tester's trajectory to those performed by the human-like agent. The lower the cross-entropy, the more similar the interactions are.

10.2.1 Game A

The grid size in Game A is 6×7. When combined, human testers could detect 90% of the defects. However, individual performance was only around half of this score, as shown in Table 9. Except for BR-MCTS, human-like MCTS agents with a 40ms computational budget were able to discover all of the faults, and these agents produced a comparable sequence length. MM-MCTS and SP-MCTS have lower cross-entropy scores than the other MCTS agents (Table 10). The computing budget increases the number of bugs found while decreasing sequence lengths. For every agent except FE-MCTS, cross-entropy scores similarly declined when we increased the computational budget. With a computational budget of 40ms, the synthetic agent could not detect all of the faults. FE-MCTS has the shortest sequence length, while SP-MCTS has the greatest bug detecting score. The increase in the computational budget has a good effect on FE-MCTS and MM-MCTS. With a computational budget of 40ms, baseline MCTS agents identified at most 44% of the flaws, while increasing the computational budget reduced the bug discovery percentage to 40%. Overall, human-like MCTS scores are comparable to human-like Sarsa(λ), while synthetic Sarsa(λ) scores are superior to synthetic MCTS.

10.2.2 Game B

The grid size in Game B is 8×9. Table 9 demonstrates that human testers discovered all bugs when their individual scores were combined. When they are evaluated separately, however, their scores are

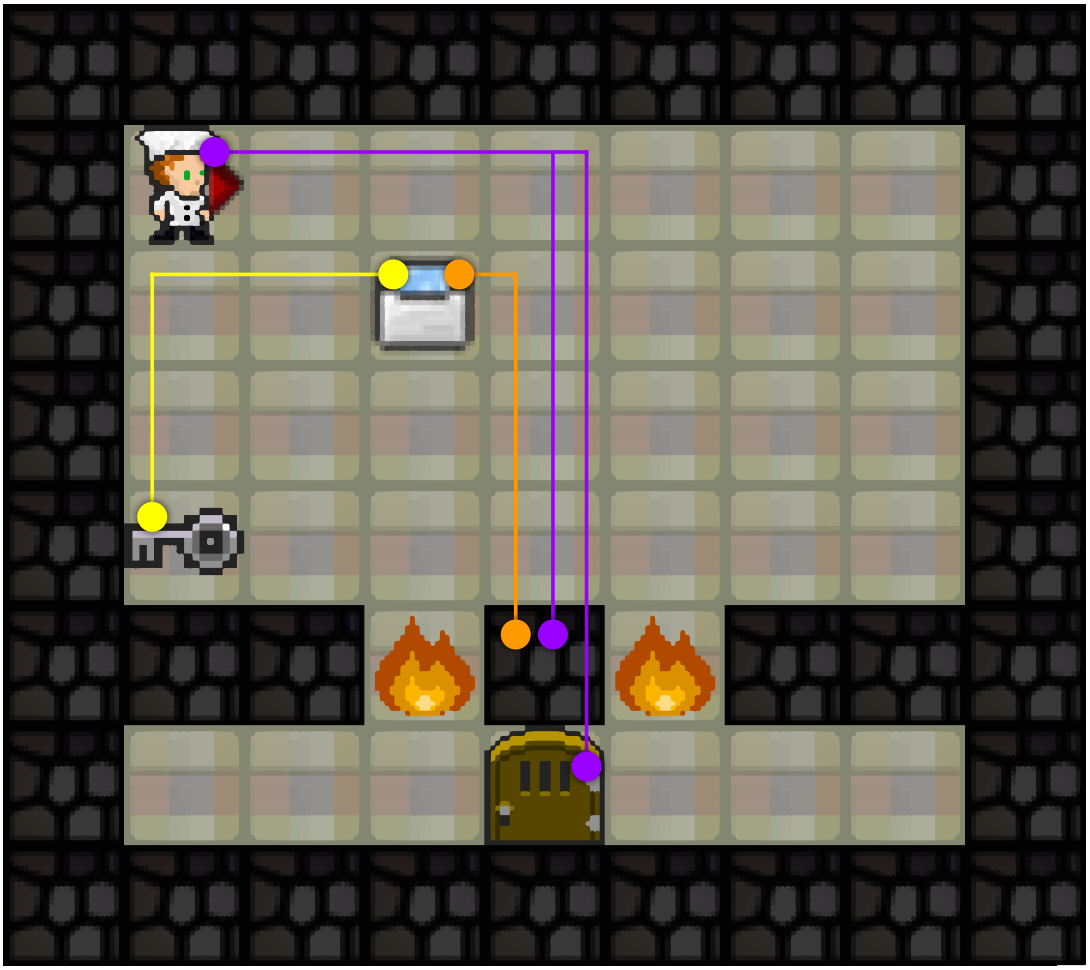


Figure 24: ©2020 IEEE.
Actions that causes faults in Level 2 of Game B.

lower than Game A. None of the agents could find all the bugs in Game B. FE-MCTS discovered more than other MCTS agents within the 40ms computational budget. Except for BR-MCTS, the sequence length of all MCTS agents is comparable. The lowest cross-entropies are found in KBE-MCTS, followed by FE-MCTS (Table 10). The cross-entropies of all agents were reduced when the computational budget was increased. This increase benefited the bug-finding performance of all agents except FE-MCTS. Synthetic FE-MCTS discovered more flaws than other synthetic MCTS agents under both computational budgets. KBE-MCTS, FE-MCTS, and BR-MCTS had similar and better baseline scores than MM-MCTS and SP-MCTS. Human-like Sarsa(λ) and synthetic Sarsa(λ) had greater bug detection rates than human-like and synthetic MCTS.

The bugs on Level 2 of Game B are represented as lines in Figure 24. Line in yellow: The two sprites overlap when the *Avatar* pushes the *Water Bucket* into the *Key*. The rule to prevent this overlap is lacking in VGDL, which is a game design flaw. Orange Line: The *Avatar* can shove the *Water Bucket* into the *Wall*. Another flaw in the game design is the lack of a collision rule between this specific *Wall* and *Water Bucket* in VGDL. Purple Line: The *Avatar* can travel through the *Wall* and complete the game without collecting the *Key*. These specifications are present in the game concept; however, they are not implemented in VGDL.

10.2.3 Game C

The grid size in Game C is 10×11, the largest of the three games. As shown in Table 9, various MCTS agents outperformed the Sarsa(λ). Human testers have the lowest individual bug detecting performance, and their combined performance is 90%. While synthetic MCTS agents do not outperform synthetic Sarsa(λ) agents, human-like MCTS agents outperform human-like Sarsa(λ) agents, and some even outperform human testers. The increase in the computational budget improves the performance of all MCTS agents except the baseline BR-MCTS. Human-like FE-MCTS and SP-MCTS outperformed all other agents with a computational budget of 300ms. MM-MCTS has the shortest trajectory among human-like agents, while FE-MCTS has the lowest cross-entropy (Table 10). Among synthetic MCTS agents, Synthetic BR-MCTS has the highest bug detection rate. In certain instances, the baseline agent could not detect any problems, while FEBR-MCTS performed the best.

10.3 Results for Developing Persona and APF

10.3.1 Experiment I: Procedural vs Goal-based personas:

The interactions of seven distinct personas are presented in Table 11. The Exit persona navigates immediately to the *Door*, located four spaces below the *Avatar*. The other three procedural personas proceed to the same *Door*, but they also gather the *Treasure* and slay the *Monster* on the way. The Developing Killer persona defeats all of the *Monsters* in the level's top half. The Developing Collector persona acquires four *Treasures* on the level's top half. The Developing Raider is a hybrid of the Developing Killer and the Developing Collector, killing *Monsters* and collecting *Treasures* in the top part of the level. Finally, the Developing Completionist kills and gathers more *Monsters* and *Treasures* than any other persona. Developing Completionist, kills all *Monsters* and collects all *Treasures*, except the *Monster* and *Treasure* below the beginning location. All procedural personas interact with a limited portion of the level, whereas developing personas engage with a larger portion. For this reason, we ran

Table 9: ©2020 IEEE.
Bug Finding Percentage and Trajectory Length of Human Testers, Sarsa(λ), and MCTS. The values shown with range have values Confidence Interval of 0.95.

Tester	Bug Finding Percentage %			Trajectory Length		
	Game A (6x7)	Game B (8x9)	Game C (10x11)	Game A (6x7)	Game B (8x9)	Game C (10x11)
Humans						
Combined	90.0	100.0	90.0	41.0 − 58.8	38.8 − 49.7	74.5 − 99.9
Individual	42.7 − 56.0	29.5 − 46.3	26.7 − 43.7			
Sarsa(λ)						
Synthetic	100.0	76.2	70.0	31.6 − 50.1	75.8 − 89.1	129.0 − 148.4
Human-Like	100.0	90.5	70.0	39.3 − 48.0	47.7 − 53.2	97.9 − 109.6
Baseline	30.0	42.9	10.0	6.8 − 14.8	13.1 − 21.8	30.2 − 65.3
Computational Budget 40ms				**Sequence Length**		
KBE-MCTS						
Synthetic	84.0 − 90.0	64.0 − 70.0	36.0 − 50.0	84.4 − 109.3	98.4 − 108.0	211.6 − 232.5
Human-Like	86.0 − 100.0	67.0 − 73.0	60.0 − 68.0	70.2 − 78.6	61.3 − 66.5	90.1 − 99.5
Baseline	20.0 − 20.0	34.0 − 41.2	10.0 − 10.0	14.7 − 38.9	52.0 − 70.7	72.9 − 129.1
MM-MCTS						
Synthetic	82.0 − 90.0	46.0 − 58.6	42.0 − 50.0	98.5 − 129.2	100.6 − 111.4	199.8 − 219.4
Human-Like	92.0 − 100.0	74.6 − 83.0	58.0 − 88.0	72.1 − 80.2	61.9 − 67.8	74.5 − 83.5
Baseline	26.0 − 40.0	9.0 − 17.0	0.0 − 0.0	19.2 − 59.0	67.3 − 104.6	91.8 − 132.6
FE-MCTS						
Synthetic	84.0 − 90.0	59.8 − 68.0	52.0 − 64.0	62.7 − 84.8	97.0 − 106.0	195.9 − 215.0
Human-Like	92.0 − 100.0	76.6 − 87.0	72.0 − 80.0	70.8 − 78.8	63.2 − 68.7	80.3 − 89.5
Baseline	20.0 − 28.0	33.0 − 39.4	10.0 − 10.0	13.2 − 32.8	58.7 − 85.1	48.0 − 102.0
BR-MCTS						
Synthetic	82.0 − 90.0	57.8 − 65.0	50.0 − 66.0	96.7 − 128.4	112.8 − 125.8	207.1 − 228.2
Human-Like	76.0 − 90.0	76.0 − 82.0	56.0 − 70.0	89.0 − 99.8	103.0 − 112.6	117.1 − 130.6
Baseline	20.0 − 20.0	32.0 − 40.4	13.2 − 26.4	22.9 − 82.6	53.5 − 82.3	36.7 − 76.4
SP-MCTS						
Synthetic	84.0 − 96.0	46.8 − 60.2	38.0 − 58.0	99.3 − 127.9	107.2 − 118.8	197.2 − 217.9
Human-Like	94.0 − 100.0	72.0 − 83.2	70.0 − 92.0	74.4 − 82.8	61.6 − 67.5	75.9 − 85.1
Baseline	26.0 − 44.0	15.0 − 19.0	0.0 − 6.0	15.5 − 35.2	65.3 − 105.5	61.3 − 130.1
Computational Budget 300ms						
KBE-MCTS						
Synthetic	84.0 − 90.0	61.0 − 71.0	46.0 − 60.0	76.5 − 97.8	103.4 − 112.9	219.9 − 239.0
Human-Like	100.0 − 100.0	75.6 − 84.0	68.0 − 90.0	63.5 − 71.1	67.9 − 73.5	109.3 − 118.5
Baseline	20.0 − 26.0	30.0 − 36.0	10.0 − 16.0	10.0 − 15.8	54.9 − 77.5	82.2 − 132.0
MM-MCTS						
Synthetic	84.0 − 96.0	49.6 − 64.0	48.0 − 70.0	86.2 − 109.1	95.9 − 106.3	206.2 − 226.9
Human-Like	100.0 − 100.0	79.4 − 87.0	68.0 − 92.0	66.0 − 73.4	64.2 − 69.7	77.7 − 87.2
Baseline	30.0 − 38.0	15.0 − 22.2	0.0 − 12.0	18.7 − 55.1	52.8 − 79.0	64.5 − 132.3
FE-MCTS						
Synthetic	90.0 − 96.0	60.6 − 74.0	50.0 − 66.0	49.6 − 64.3	88.8 − 97.2	202.8 − 220.0
Human-Like	94.0 − 100.0	76.8 − 82.2	76.0 − 94.0	47.4 − 52.8	53.4 − 57.8	105.7 − 115.0
Baseline	24.0 − 40.0	35.8 − 42.0	10.0 − 10.0	10.8 − 18.1	38.4 − 55.9	89.7 − 131.6
BR-MCTS						
Synthetic	80.0 − 88.0	66.0 − 72.0	60.0 − 66.0	61.7 − 85.3	77.7 − 86.8	198.8 − 218.4
Human-Like	84.0 − 90.0	76.8 − 82.2	74.0 − 80.0	77.0 − 85.6	88.7 − 97.0	116.6 − 128.4
Baseline	22.0 − 34.0	21.4 − 32.2	2.0 − 14.0	15.1 − 31.4	42.4 − 64.1	54.4 − 117.4
SP-MCTS						
Synthetic	84.0 − 96.0	44.0 − 58.4	46.0 − 62.0	83.5 − 108.4	105.1 − 115.7	202.4 − 223.8
Human-Like	100.0 − 100.0	81.0 − 85.0	76.0 − 94.0	68.5 − 75.9	63.9 − 69.3	82.5 − 91.6
Baseline	30.0 − 38.0	15.0 − 20.6	0.0 − 6.0	16.4 − 42.8	48.9 − 76.1	57.2 − 130.6

Table 10: ©2020 IEEE.
Cross-Entropy Results of Sarsa(λ) and MCTS. The values shown with range have values Confidence Interval of 0.95.

Human-like Agent	Cross-Entropy		
	Game A (6x7)	Game B (8x9)	Game C (10x11)
Sarsa(λ)	$0.27 - 0.37$	$0.57 - 0.69$	$0.69 - 0.82$
Computational Budget 40ms			
KBE-MCTS	$0.72 - 0.82$	$1.12 - 1.20$	$1.15 - 1.22$
MM-MCTS	$0.57 - 0.64$	$1.24 - 1.33$	$1.20 - 1.27$
FE-MCTS	$0.61 - 0.70$	$1.16 - 1.25$	$1.18 - 1.25$
BR-MCTS	$1.10 - 1.22$	$1.22 - 1.31$	$1.15 - 1.22$
SP-MCTS	$0.58 - 0.66$	$1.21 - 1.30$	$1.20 - 1.27$
Computational Budget 300ms			
KBE-MCTS	$0.65 - 0.75$	$1.00 - 1.07$	$1.05 - 1.12$
MM-MCTS	$0.57 - 0.65$	$1.11 - 1.20$	$1.21 - 1.28$
FE-MCTS	$0.75 - 0.85$	$0.98 - 1.06$	$0.98 - 1.05$
BR-MCTS	$0.93 - 1.05$	$1.00 - 1.08$	$1.08 - 1.15$
SP-MCTS	$0.53 - 0.61$	$1.12 - 1.20$	$1.19 - 1.26$

the same experiment with PPO + CTS RL agent for procedural personalities. The interactions done by procedural personas as the agent explores the environment are shown in Table 11. We can see that the interactions done by the PPO + CTS RL agent are more in line with the persona's decision model.

Table 11: ©2022 IEEE.
Interactions performed by the PPO RL agent in Experiment I.

	Game Event		
Personas	Monsters Killed	Treasures Collected	Door
Exit	0	0	1
Monster Killer	1	1	1
Treasure Collector	1	1	1
Completionist	1	1	1
Dev. Killer	3	0	1
Dev. Collector	1	4	1
Dev. Raider	3	4	1
Dev. Completionist	5	8	1

10.3.2 Experiment II: Alternative paths found in GVG-AI:

We used the path found by the Exit persona in Experiment I to train APFCTS (see Path 1 in Figure 25). Then, we trained the PPO + CTS + APFCTS agent in the first testbed game while using the Exit persona's utility weights. We repeated the experiment for each path obtained from the PPO + CTS + APFCTS agent. First, an APFCTS is trained using one of the obtained paths, and then we use

Table 12: ©2022 IEEE.
Interactions performed by the PPO + CTS RL agent in Experiment I.

	Game Event		
Personas	Monsters Killed	Treasures Collected	Door
Monster Killer	2	0	1
Treasure Collector	0	3	1
Completionist	2	3	1

this trained APFCTS to train a PPO + CTS + APFCTS agent. The paths identified at the end of the process are shown in Figure 25. Table 13 shows the total discounted rewards—the rewards received from the environment and the APFCTS. The bold values indicate the alternative paths of the trained path. For example, Path 1 has four alternative paths—Paths 2 to 6. Table 13 also shows that, when we use APFCTS, the reward of playing the same path decreases by at least 0.1, and the reward of space-disjoint paths increases by at least 0.1. This reward difference justifies why APF supports finding alternative paths.

Lastly, Table 13 shows that APFCTS clusters the paths in Experiment II into two equivalence classes, which are $\{1, 2\}$ and $\{3, 4, 5, 6\}$. Therefore, we may interpret that distinct paths refer to *paths that are space-disjoint from the one trained on* for APFCTS.

Table 13: ©2022 IEEE.
Total Discounted Reward without APFCTS and with APFCTS. The first row shows the total discounted reward without APFCTS. For the rows with a path number, the number indicates which path we used to train the APFCTS. The values under tested paths show the total discounted reward that the agent receives when APFCTS modulates the environment reward. The bold values demonstrate the found paths when we execute the PPO + CTS + APFCTS agent.

	Tested Paths					
Trained Path	1	2	3	4	5	6
-	**0.86**	0.78	0.84	0.84	**0.86**	0.76
1	0.76	0.77	**0.98**	**0.98**	**0.98**	**0.98**
2	0.82	0.61	**0.98**	**0.99**	**0.98**	**0.98**
3	**0.98**	**0.98**	0.74	0.86	0.82	0.88
4	**0.99**	**0.98**	0.86	0.72	0.86	0.86
5	**0.98**	**0.98**	0.81	0.85	0.76	0.87
6	**0.99**	**0.98**	0.87	0.87	0.88	0.60

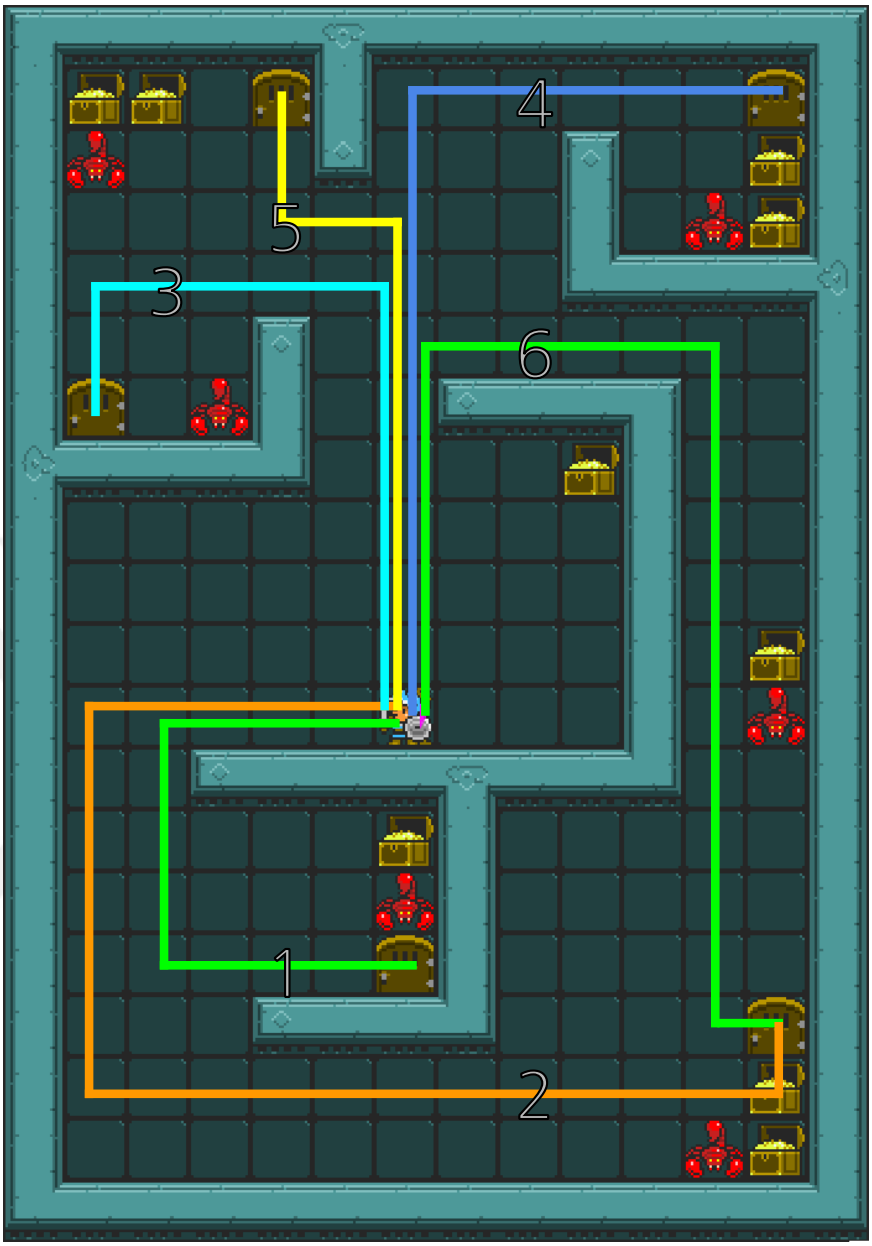


Figure 25: ©2022 IEEE.
Paths found by Exit persona with PPO and with PPO + CTS + APFCTS.

10.3.3 Experiment III: Personas in Doom:

In the second testbed game, a Doom level, we tested with nine various personalities (see Figure 17). Table 14 displays the interaction results, and all personas react following their specifications. The Exit persona always completes the game, and rarely kills a *Monster* but never obtains a *Treasure*. The Monster Killer persona typically kills all *Monsters*, seldom obtains *Treasures*, and always completes the game. Monster Killer is identical to Developing Killer. However, it only kills half of the *Monsters* and seldom dies. Both the Treasure Collector and the Developing Collector have similarities. They

each gather a single *Treasure*, slay the fewest *Monsters*, and perish the most. Minor distinctions exist between the Completionist, Developing Completionist, and Developing Casual Completionist personalities. The Developing Casual Completionist always completes the level but can sometimes not acquire the second *Treasure*. The Completionist and Developing Completionist routinely gather the second *Treasure*, but they seldom die and cannot complete the level.

Table 14: ©2022 IEEE.
Interactions of Personas in Experiment III over 1000 evaluations.

	Game Event			
Personas	Monsters	Treasures	Door	Death
Exit	0.27 ± 0.48	0.00 ± 0.00	1.00 ± 0.00	0.00 ± 0.00
MK	5.79 ± 0.91	0.01 ± 0.07	0.98 ± 0.15	0.00 ± 0.00
Dev. Killer	3.54 ± 0.98	0.01 ± 0.08	0.96 ± 0.19	0.01 ± 0.08
TC	1.94 ± 0.70	0.94 ± 0.24	0.80 ± 0.40	0.19 ± 0.39
Dev. Collector	2.00 ± 0.65	0.95 ± 0.22	0.87 ± 0.34	0.13 ± 0.34
Dev. Raider	3.52 ± 0.73	0.98 ± 0.15	0.97 ± 0.17	0.01 ± 0.08
Comp.	5.76 ± 1.06	1.91 ± 0.38	0.95 ± 0.22	0.01 ± 0.11
Dev. Comp.	5.81 ± 0.92	1.91 ± 0.36	0.96 ± 0.19	0.01 ± 0.09
Dev. Cas. Comp.	5.83 ± 0.53	0.98 ± 0.13	0.98 ± 0.14	0.00 ± 0.00

10.3.4 Experiment IV: Alternative paths found in Doom:

In the third testbed game, we used the PPO + ICM agent to train an Exit persona. The first path indicated in Figure 26 is the Exit persona's trajectory. Using this initial approach, we trained an APFICM, and then we trained a new Exit persona using a PPO + ICM + APFICM agent. The new Exit persona took the second route. Table 15 shows the total discounted reward collected by these two Exit personas. Because the first road has 52 steps and the second path has 77 steps, the total reward of the first path is more than that of the second. APFICM, on the other hand, boosts the total reward from the second path while lowering the total reward from the first path.

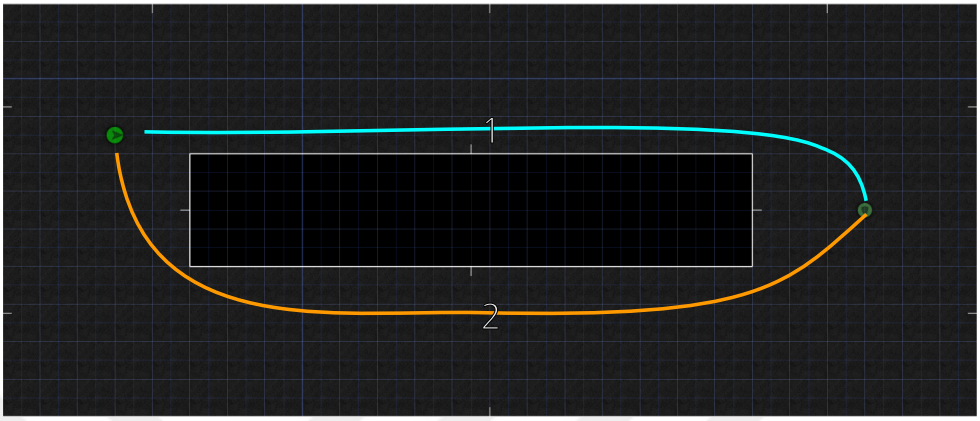


Figure 26: ©2022 IEEE.
Paths found by Exit persona with PPO and with PPO + ICM + APFICM.

Table 15: ©2022 IEEE.
Total Discounted Reward without APFICM and with APFICM over 1000 evaluations. First row shows the total discounted reward without APFICM. For the rows with a number, we train the APFICM and calculate the discounted reward by using ICM. The bold values demonstrate the found paths when we execute the PPO + ICM + APFICM agent by training APFICM with the trained path.

	Tested Paths	
Trained Path	1	2
-	**0.80** $\pm$ **0.02**	0.68 $\pm$ 0.01
1	0.51 $\pm$ 0.02	**0.78** $\pm$ **0.01**

CHAPTER 11

DISCUSSION

In this thesis, we first established a technique for capturing tester behavior, known as test state. We then introduced two strategies for generating test goals for agents: synthetic and human-like. We used Sarsa(λ) and MCTS RL agents to achieve these test objectives. Next, we analyzed the agents' bug-finding abilities in three games and assessed how similar the human-like agents were to the original human testers. Following that, we experimented with various MCTS modifications in order to produce a better tester agent. We tested these modifications on three games with 45 bugs, evaluating their effects on bug-finding performance, how they are affected by the computational budget, and comparing these results with human testers and an agent using Sarsa(λ). Furthermore, we highlighted an enhancement for procedural persona, developing persona, to allow game creators to playtest their games effectively. Finally, we presented APF, a strategy for empowering RL agents to discover new paths. We experimented with developing persona and APF in GVG-AI and Doom environments.

Test state helped to distinguish previously equivalent states. Test state supports this behavior and many more such as attacking walls and covering all empty spaces. Consequently, we were able to model the testing behavior using MDP and MGP-IRL could learn tester heuristics from collected trajectories. Note that, these trajectories were collected from games that contained bugs. Though we used the test state primarily for testing, it can benefit the gameplay. There can be many hidden doors and other rare objectives in a game, and an agent utilizing the test state can engage with these objectives.

Creating a synthetic test objective from the game scenario graph and modifying it was beneficial because the synthetic agent outperformed the baseline in all experiments (Figure 19). In Game B, which features the most complicated game scenario graph, the baseline agent outperformed half of the individual human testers (Figure 19 and Figure 20). Because the underlying graph information is challenging to grasp, these testers could not cover all proposed pathways of the game. Modifications led the agent to various pathways, while graph coverage gave the path for playing the game. The game graph has significant advantages since it forces the agent to play each desired scenario and ensures that these routes are covered, unlike in [71]. Furthermore, the modifications urged the agent to emphasize the game's constraints. In the experiments, the synthetic agent outperforms every individual tester in human-like agents and the majority of human testers (Figure 19 and Figure 20), demonstrating the synthetic agent's potency. Finally, the synthetic agent offers a customizable approach with a configurable coverage requirement and various adjustments for conducting tests without gathering massive data. The Sarsa(λ) agent, in contrast to the MCTS agent, used synthetic test objectives better since it discovered more problems in all three games (Figure 19).

Human testers could neither identify the defects nor outperform the synthetic agent individually. However, when their performance results were combined, they could find most bugs and outperform the synthetic agent in two of three tests (Figure 19). This situation is also apparent in human-like testers, where the combined human-like agent outperformed the synthetic agent in every trial (Figure 19). Several human testers went through various game pathways and found various bugs. As a result, finding distinct testers is critical. Furthermore, because human-like test goals were generated from these testers' sequences, they also profited from this variation. When we examine the bug-detecting performance of different human-like agents, we find that the human-like Sarsa(λ) agent with a probability threshold of 0.0 leads in both individuals (Figure 20) and overall (Figure 19) performance. This achievement can be attributed to the multi-goal strategy. First, a simple goal is easier to play than a complicated one; second, when an agent plays a different level, verifying one goal at a time is best since level composition may induce the agent to skip a feature too soon. Third, the test steps must be carried out in the correct order.

MGP-IRL was proposed to generate human-like test goals based on collected tester trajectories. All of the tests in Section 9.1 show that human-like Sarsa(λ) agents with a probability threshold of 0.0 could perform more comparable interactions to human testers (see Figure 21). We discovered that when the probability threshold grows, so does the mean cross-entropy. The bug-finding performance of these agents follows a non-increasing trend when the probability threshold is increased (Figure 20). As a result, the human-like agent with a probability threshold of 0.0 is both the most human-like and the most successful in discovering bugs. The number of times the MGP-IRL divided the trajectory is shown in Figure 23. In all games, the likelihood threshold of 1.0 extracted the weights of the features using the whole trajectories.

We employed the Sarsa(λ) and MCTS agents to construct test sequences. The agents completed the test goals after receiving rewards from various interactions. The achievement of test objectives is reviewed using criteria, and if the agent meets the criteria, the agent moves on to the next test goal in the sequence. This strategy led our agent to investigate numerous test objectives. Sarsa(λ) outperforms MCTS in terms of mean bug-detecting performance (Figure 19). Among all agents, the synthetic MCTS agent discovered the fewest bugs. Our manually designed weights were better suited to the Sarsa(λ) agent than the MCTS agent. On the other hand, one of the human-like MCTS agents with a probability threshold of 0.0 was able to detect all of the bugs, which was not the case for this agent when Sarsa(λ) was used (Figure 20 Game A). After a thorough investigation, we discovered that some fake walls caused the discrepancy. Because the human agent did not check all of the barriers, the MCTS agent uncovered this flaw due to the stochastic nature of MCTS. Furthermore, when we aggregated the unique bugs detected in all MCTS runs, we discovered that human-like MCTS agents could find 90% of the bugs in Game C, which is the same as humans'. As a result, we conclude that the stochasticity of MCTS is advantageous in testing.

MCTS and Sarsa(λ) come equipped with built-in procedures for finding bugs, provided the agent is given the necessary objectives and features. In the first experiment (Figure 19 Game A), the baseline agent surpassed one of the human testers owing to the exploration aspect of these techniques. If the agent's purpose is to obtain the key and it is only feasible after attacking the door due to a flaw, then the agent may find this precise sequence. Trajectory plots reveal a difference between testing and game playing —performed by the baseline agent. Game A features a modest board, yet human testers performed over 100 actions (see Figure 22 Game A). In the same game, the baseline agent may only do 15 actions, which is the length of the game scenario graph's route. MCTS agents performed more actions than Sarsa(λ) agents in all experiments described in Section 9.1 (see Figure 22). This difference

is expected because Sarsa(λ) optimizes the whole sequence, whereas MCTS has to choose an action in 300ms. As a result, the MCTS cross-entropy values are lower in all three experiments (see Figure 21). The distinction between testing and gameplay also appears in other games (see Figure 22). Another notable difference is that the shape of the synthetic agent differs much from that of human testers, but the shape of human-like testers mimics that of human testers (see Figure 22). The non-human nature of our synthesized method is also shown in this way.

We investigated many MCTS modifications to increase an MCTS agent's game testing performance. In addition, we investigated the impact of the computational budget on the MCTS modifications. Our experiments in Section 9.2 for human-like agents show that increasing the computational budget has a favorable effect on the majority of the MCTS agents. KBE-MCTS gains the most from this increase. As a result, we may conclude that KBE-MCTS could not explore the tree with a computational budget of 40ms. SP-MCTS and MM-MCTS are also favorably influenced, but not as much as KBE-MCTS. Furthermore, when more computation is done, they become more stable agents, and as a result, their confidence interval narrows. BR-MCTS outperforms KBE-MCTS in Game B and Game C with a 40ms computational budget. The increase in the computational budget, on the other hand, reverses the situation. The bias in BR-MCTS rollouts restricts its upper bound and makes it the most stable agent. The only advantage of FE-MCTS over KBE-MCTS is the ability to reuse trees. Tree reuse improves the bug-finding percentage significantly under 40ms, but KBE-MCTS outperforms FE-MCTS in Games A and B as the computational budget increases. Because these games are small compared to Game C, tree reuse reduces stochasticity. FE-MCTS also performs the shortest sequences within a computational budget of 300ms. The increased computational budget for synthetic agents improved the bug detection performance of all agents, indicating that synthetic test objectives are more difficult to achieve than human-like test goals. A baseline agent's ability to identify flaws is restricted to the bugs in the scenario, and baseline Sarsa(λ) indicates this proportion. Therefore, the increase in the computational budget had a beneficial effect on the baseline FE-MCTS. The impact of the computational budget is unclear for other baseline MCTS agents. In addition, there are cases where a baseline MCTS agent outperforms the baseline Sarsa(λ). This increase in bug-finding performance is also due to the stochasticity of the MCTS, which also helped to discover the fake walls in [52].

For bug detection, we proposed a set of six possible modifications to the MCTS agent. SP-MCTS, with a computational budget of 300ms, is the best overall for human-like agents. Although FE-MCTS may approach and even exceed the same upper bound, FE-MCTS achieves this performance when both computational budgets are considered. In these cases, however, SP-MCTS sequence lengths are shorter than FE-MCTS. The MixMax modification can explain this variation. MM-MCTS and SP-MCTS have similar upper bounds, but because SP-MCTS explores more, it assures a better lower bound. The least effective human-like agent, BR-MCTS, is reliable. When it comes to synthetic agents, SP-MCTS is one of the least successful, whereas BR-MCTS has begun to thrive. This shows that synthetic test goals are positioned further down in the tree than human-like objectives. As a result, for synthetic test goals, MixMax, and Tree Reuse is effective. BR-MCTS is also valuable because these objectives can be discovered through biased rollouts. In synthetic MCTS, there is no obvious winner; however, FE-MCTS looks promising.

Furthermore, when we compare the MCTS variants with Sarsa(λ) using Table 9. In Game A, human-like agents using MCTS variants reach the bug finding percentage obtained with Sarsa(λ), and they can achieve this with a 40ms computational budget, except BR-MCTS. In Game B, the human-like agent using MM-MCTS, FE-MCTS, and SP-MCTS is 3-5% behind of Sarsa(λ). In Game C, the MM-MCTS, FE-MCTS, and SP-MCTS with 40ms computational budget and all human-like MCTS agents

with 300ms surpass Sarsa(λ)'s bug-finding performance. These bug finding percentages show that human-like MCTS agents can compete with Sarsa(λ) in bug-finding with the advantage of using a less computational budget. For synthetic agents, we observe that bug finding performance of Sarsa(λ) is better. Nevertheless, in Game A and Game B, FE-MCTS; in Game C, MM-MCTS are the best competitors. Furthermore, for every game, we can see that Sarsa(λ) produces shorter sequences, which are more human-like than MCTS agents. For the baseline agent, FE-MCTS with 300ms computational budget performs closest to Sarsa(λ), thanks to the tree reuse modification.

MCTS, on the other hand, may perform more random actions than Sarsa(λ). This behavior has an impact on the agent's human-likeness. When we compare the cross-entropy scores in Table 10, we see a direct relationship between the computation budget and the human-likeness of KBE-MCTS, SP-MCTS, and MM-MCTS, but not for FE-MCTS and MM-MCTS. However, we cannot claim that an agent would uncover more bugs if it behaves more like the original person. Human testers' heuristics offer the goals for testing the game, and any randomness added while constructing a sequence reduces the similarity. Furthermore, owing to randomness, multiple runs may discover different bugs.

Game designers employ procedural personas and developing personas to automate the playtesting process. One disadvantage of procedural personas stems from the utility function. A utility function implements a persona's decision model. A Treasure Collector, for example, receives positive feedback after completing the level and collecting a *Treasure*. However, if the agent's starting location is near the *Door*, the agent may overlook the *Treasures*. On the other hand, if the *Door* is placed after the *Treasures*, the agent is more likely to engage with the majority of the *Treasures*. This dilemma was demonstrated in Experiment I of Section 9.3. The procedural personas Monster Killer, Treasure Collector, and Completionist all performed the identical set of activities in the absence of any exploring technique. When we included the exploration technique in the agents who realize these personas, the set of acts performed by these personas changed. Additionally, these new sets of activities matched their decision model better. This issue is also seen in the MCTS agent playtesting of the MiniDungeons 2 game [36].

The issue with the utility function is that it is an aggregation of multiple goals. As a result, depending on the level composition, and the hyperparameters of the RL agent, the procedural persona reflects only one of those playstyles. We feel that the Developing Completionist matches the notion of a "Completionist" character better than the procedural Completionist in Experiment I in Section 9.3. Developing personas tackles this issue by offering a series of goals. As a result, a game designer may utilize the developing persona to select the playstyle carefully she wishes to playtest.

Developing personas have the added benefit of supporting changing playstyles, which gives them an edge over procedural personas. For example, in Experiment I of Section 9.3, the Developing Raider killed the *Monsters* and then collected the *Treasures*. The Developing Raider begins the game as a Monster Killer and, after meeting a criterion, transforms into a different persona —a Treasure Collector—. These development sequences were discussed by Bartle [10], but they were impossible with a single utility function. As a result, this behavior displayed by Developing Raider was absent in procedural personalities. Another critical feature of playtesting is the capacity to construct play traces as though humans generated them. We used handcrafted utility functions in this study; however, these utility functions could have been extracted from human playtest data using Inverse Reinforcement Learning [89]. This modification may aid the RL agent in producing a more human-like playtest [5, 92].

We conducted experiments on the Doom environment in addition to the GVG-AI environment (see Section 9.3). We believe that our study is the first to playtest personas in a 3D environment. The researchers [36, 56] used the MCTS RL agent to realize personas in 2D surroundings. Unfortunately, MCTS would be a poor choice for 3D environments, and MCTS would underrepresent the persona. As a result, we employed the PPO agent in Experiments III and IV in Section 9.3 because PPO is a competent agent used by OpenAI [8]. Experiment III in Section 9.3 shows that the PPO agent correctly realized the decision models of personas. We deduce that a player must kill a *Monster* to complete this level based on the findings in Table 14. The level is the most difficult for Treasure Collector and Developing Collector because they must slay a *Monster* to gather the *Treasures*. When we compare the Developing Casual Completionist and Developing Completionist personas, we discover an essential aspect of the game. The former never dies but only collects one *Treasure*, while the latter rarely dies but gathers both *Treasures*. Based on this information, we may conclude that obtaining the second *Treasure* results in the player's death. Because the Developing Casual Completionist is more concerned about losing her health, she considers collecting the second *Treasure* dangerous. Furthermore, when the Killer and Developing Killer personas are compared, the latter dies more than the former. This comparison reveals another aspect of this level. If a player enters a fight to kill *Monsters*, he or she should kill as many as possible. Otherwise, this player, like the Developing Killer, will die. On the other hand, the Developing Casual Completionist kills as many *Monsters* as a Monster Killer. This indifference suggests that the game may not be demanding enough for a hardcore gamer.

Limitations & Challenges: The test state increased the number of states explored by our agents, which creates an issue for tabular RL. However, the test state is considerably simple and requires less than 2KB of space for Game C. Nevertheless, the solution for this problem is to use RL agents that use function approximators such as DQN [55] or PPO [83].

The MGP-IRL algorithm is the most crucial component influencing human-like performance. We can improve its greedy approach with dynamic programming, but this substitution will increase the time required to develop a test objective. On the other side, we can instruct a human tester to test a game, allow the tester to segment the trajectory, and then use MGP-IRL to determine the feature's repeat count, preferred direction, and rewards. This strategy, however, will apply to internal testing rather than open beta testing. It is also worth noting that the weights suggested by IRL stabilize with the quantity of data it analyses. This method requires that the tester repeats the same objective in several runs.

Furthermore, if a tester discovers a bug, she will exploit it. When MGP-IRL attempts to structure these trajectories, the exploited sequences are generalized to all scenarios. This generalization makes it impossible to understand the tester's actions. For example, if a tester discovers a wall through which the avatar may pass, the tester tries to carry other objects, but human-like agents interpret this wall as any other wall. As a result, the agent will try to engage with any wall. MPG-IRL generalizes the fake wall as any wall. Consequently, the test goal extracted from this sequence will be ill-formed. Furthermore, unlike Game B, Game C includes a free puzzle; thus, testers had different solutions to this puzzle. Therefore, this behavior was neither easily caught nor easily duplicated.

We chose movable sprites over enemies because tester agents would check whether an enemy's interactions are correct with different sprites. Since the enemies act randomly, to observe the same interactions, the tester agents should restart the game until they observe the desired behavior. Nevertheless, this behavior is not relatable to a human tester.

However, all of our proposed methods rely on RL algorithms. As a result, if these agents cannot play a game, we cannot utilize RL agents to test this game. Finally, APF performance is related to the success of exploratory strategies. We encountered this issue while using Doom in our experiments. Doom's frames confused the CTS algorithm, causing it to fail to discriminate between states. However, ICM successfully identified the states in Doom; consequently, we used APFICM in Doom. Nonetheless, there may be scenarios where ICM fails, and we must employ an alternative exploration algorithm.

Finally, testing is complex for RL algorithms because the environment does not provide a clear goal compared to the gameplay. As a result, the creation of test objectives becomes the responsibility of game designers.

CHAPTER 12

CONCLUSION & FUTURE WORK

This thesis focused on the problem of creating intelligent agents that play and test video games. We presented three approaches to generating goals: developing persona, synthetic, and human-like agent. Next, we employed RL agents to realize these goals on GVGAI and VizDoom environments. Furthermore, we introduced new methodologies to improve the testing capabilities of RL agents. For MCTS tester agents, we compared several modifications while presenting a novel modification. Lastly, we introduced Alternative Path Finder, which helps agents to discover unique ways of playing and testing a game. We submitted all of the approaches mentioned in this thesis and published two journals and one conference paper.

In our first paper, we were interested in the problem of creating tester agents. The game-playing RL agents demonstrated that in arcade games [55], Go [85], StarCraft II [96], and Dota 2 [60], RL agents can surpass humans. However, game testing agents were outdated compared to game playing agents. We proposed a test state to capture and execute tester behavior in this regard. Additionally, we provided two methods for creating test objectives: synthetic and human-like. The game scenario graph served as the foundation for the synthetic test objectives. We further modified these objectives to investigate the consequences of unanticipated game transitions. Using MGP-IRL, human testers' acquired trajectory data is used to understand human-like test objectives. MGP-IRL extracted the tester heuristic as attributes, which were then transformed into objectives. To play these test objectives, we used MCTS and Sarsa agents. The agent was driven to various game states by these objectives, and the agent produced test sequences. In order to assess whether the game acts as predicted, we executed these sequences, and the automated test oracle verified the states and actions.

In the GVGAI environment, we tested with MCTS and Sarsa agents. Our findings demonstrated that the test state aided in capturing the human tester heuristic even when the game contained faults and allowed MCTS and Sarsa agents to play the game as testers. The synthetic agent outperformed the baseline agent and the vast majority of individual human-testers and human-like agents. Furthermore, when individual human-like agents work together, they may compete with the performance of synthetic agents and human testers. We also looked into the bug detection capabilities of MCTS and Sarsa agents. We discovered that Sarsa's mean performance is better than MCTS's. However, we also discovered that MCTS's stochasticity is advantageous in game testing. Furthermore, the cumulative scores of all five MCTS run demonstrated that the human-like MCTS agent competes with humans. Finally, the human-like agent with a likelihood threshold of 0.0 behaved similarly to the human testers, thanks to our MGP-IRL method. We showed that these agents could successfully test unexplored levels. Furthermore, the synthetic agent extends model-based testing by bringing the widely accepted

agent notion in gaming to traditional test approaches. Additionally, once constructed, these bots can test a game unlimited times, reducing the need for human testing.

In our experiments, the Sarsa agent outperformed the MCTS agent in game testing. We proposed several modifications to improve the bug-finding capabilities of the MCTS agent. We looked at how these modifications affected bug finding capabilities of MCTS agents. In this regard, we proposed using transpositions, knowledge-based evaluations, tree reuse, MixMax, Boltzmann rollouts, and SP-MCTS. We tested these modifications in three GVGAI games. Our synthetic and human-like test goals were executed using MCTS to generate sequences later replayed in the game to check for problems with our oracle. We used two alternative computational budgets, 40 and 300 milliseconds, to better understand the effect of timing on these modifications. Our findings indicate that the modifications are beneficial, but their effectiveness depends on the agent type. Our experiments revealed that MM-MCTS and FE-MCTS performed better for the synthetic agent, although BR-MCTS had a better lower bound score with a shorter sequence. For human-like agents, SP-MCTS outperformed FE-MCTS within both computational budgets. Lastly, the FE-MCTS with a computational budget of 300ms performed better than the other baseline MCTS.

Additionally, our most recent article proposed creating personas to enhance game playtesting. The procedural persona approach [36, 56], which is frequently used while playtesting video games, was replaced by our developing persona proposal. With the introduction of a multi-goal technique, which served as the foundation for our developing persona, we could create different player archetypes that were not conceivable with procedural persona. We contrasted procedural and developing persona in GVGAI and Doom, two distinct environments. Our findings indicate that developing persona gives the game designers access to more in-depth and distinctive game information. As a result, we concluded that game designers might use developing personas to learn more about the game during playtesting. Finally, we demonstrated how state-of-the-art RL algorithms might be used to expand automated playtesting to 3D environments.

On the other hand, even though we developed several novel methods for playtesting and game testing, all of our methods were based on RL algorithms. The objective of RL algorithms is to determine the optimum course of action given a state; consequently, the RL algorithms are quite likely to produce the same trajectories. However, human playtesters might attempt other paths after testing a certain trajectory. The RL literature has progressed from DQN [55] to PPO [83], and agents were able to overcome humans in games that were previously considered to be impossible [85, 96, 60]. However, as humans, we usually play games for amusement, and we may even play some of these games several times. When we play these games again, we are aware of our previous strategies, so we strive to play differently to enjoy ourselves. However, advancements in RL algorithms ignore this aspect of human gameplay. We proposed the Alternative Path Finder to allow the agents to find alternative ways to play the same game. We tested APF in GVGAI and Doom environments and demonstrated that it ensures the discovery of a unique path. APF might be used for playing and testing games, even though we only tested it for playtesting. We see this as one of our most significant contributions to RL.

In the future, we may utilize a PPO agent to play our synthetic and human-like test objectives and APF to expand state coverage. Furthermore, we may extend our game testing to 3D environments using Doom [44] and Minecraft [43] environments. This future work will incorporate the methodologies we proposed in this thesis. Additionally, we might utilize a neural network to record which actions are performed rather than recording them on a test state. This information might be stored using an architecture similar to APFICM. The advantage of this enhancement is that it allows the test

state to be scaled to more complicated games. Furthermore, compared to typical game-playing trajectories, testers' test trajectories are more complicated and chaotic. Because of this, we want to make our MGP-IRL more resistant to random actions and enhance the internal IRL algorithm with more sample-efficient approaches [84]. Lastly, in the Doom environment, we used an LSTM layer in PPO architecture to play in a POMDP. Therefore, there is a possible improvement for APFICM by substituting the linear layer with an LSTM layer. This substitution will help APFICM to distinguish actions selected in succession. As a result, APFICM will learn to separate paths rather than separate actions.

Additionally, we rely on RL and exploration methods to solve the task of testing the game. However, depending on the chosen environment, these methods may fall short or use a larger computational budget to realize the given goal. We can amend this problem by utilizing curriculum learning [57]. In curriculum learning, a set of mini-tasks are generated in sequence, and the agent is trained with these mini-tasks. Afterward, the knowledge learned from these mini-tasks is transferred to the actual task. Furthermore, we can use an automated goal generation framework [33] to generate mini-goals that guide the agent in the environment. These mini-goals have an appropriate difficulty; therefore, the agent's policy is improved.

REFERENCES

[1] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 1–, New York, NY, USA, 2004. ACM.

[2] E. Adams. *Fundamentals of Game Design*. Voices that matter. New Riders, 2014.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[4] S. Ariyurek, A. Betin-Can, and E. Surer. Enhancing the monte carlo tree search algorithm for video game testing, 2020.

[5] S. Ariyurek, A. Betin-Can, and E. Surer. Automated video game testing using synthetic and humanlike agents. *IEEE Transactions on Games*, 13(1):50–67, 2021.

[6] S. Ariyurek, E. Surer, and A. Betin-Can. Playtesting: What is beyond personas. *IEEE Transactions on Games*, pages 1–1, 2022.

[7] M. Babeş-Vroman, V. Marivate, K. Subramanian, and M. Littman. Apprenticeship learning about multiple intentions. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 897–904, USA, 2011. Omnipress.

[8] B. Baker, I. Kanitscheider, T. M. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch. Emergent tool use from multi-agent autocurricula. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[9] R. Bartle. Virtual worlds: Why people play. *Massively Multiplayer Game Development 2*, 2:3–18, 01 2005.

[10] R. A. Bartle. Hearts, clubs, diamonds, spades: Players who suit MUDs. `http://www.mud.co.uk/richard/hcds.htm`, 2019.

[11] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1471–1479, 2016.

[12] M. G. Bellemare, J. Veness, and E. Talvitie. Skip context tree switching. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 1458–1466, 2014.

[13] J. A. Brown. Towards better personas in gaming : Contract based expert systems. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 540–541, 2015.

[14] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[15] Y. Burda, H. Edwards, A. J. Storkey, and O. Klimov. Exploration by random network distillation. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[16] C. Chambers, W.-c. Feng, S. Sahu, and D. Saha. Measurement-based characterization of a collection of on-line games. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC '05, page 1, USA, 2005. USENIX Association.

[17] K. Chang, B. Aytemiz, and A. M. Smith. Reveal-more: Amplifying human effort in quality assurance testing using automated exploration. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[18] B. E. Childs, J. H. Brodeur, and L. Kocsis. Transpositions and move groups in monte carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 389–395, Dec 2008.

[19] C. Cho, D. Lee, K. Sohn, C. Park, and J. Kang. Scenario-based approach for blackbox load testing of online game servers. In *2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 259–265, Oct 2010.

[20] J. S. B. Choe and J. Kim. Enhancing monte carlo tree search for playing hearthstone. In *2019 IEEE Conference on Games (CoG)*, pages 1–7, Aug 2019.

[21] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen. Ai-based playtesting of contemporary board games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, page 13. ACM, 2017.

[22] S. Devlin, A. Anspoka, N. Sephton, P. Cowling, and J. Rollason. Combining gameplay data with monte carlo tree search to emulate human play, 2016.

[23] M. S. Dobre and A. Lascarides. Online learning and mining human play in complex games. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 60–67, Aug 2015.

[24] H. Finnsson and Y. Björnsson. Learning simulation control in general game-playing agents. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[25] F. Frydenberg, K. R. Andersen, S. Risi, and J. Togelius. Investigating mcts modifications in general video game playing. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 107–113, Aug 2015.

[26] J. Fu, J. D. Co-Reyes, and S. Levine. EX2: exploration with exemplar models for deep reinforcement learning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 2577–2587, 2017.

[27] R. D. Gaina, S. Devlin, S. M. Lucas, and D. Perez-Liebana. Rolling horizon evolutionary algorithms for general video game playing. *IEEE Transactions on Games*, 14(2):232–242, 2022.

[28] F. G. Glavin and M. G. Madden. Adaptive shooting for bots in first person shooter games using reinforcement learning. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2):180–192, June 2015.

[29] A. Goldwaser and M. Thielscher. Deep reinforcement learning for general game playing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1701–1708, Apr. 2020.

[30] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén. Improving playtesting coverage via curiosity driven reinforcement learning agents. In *2021 IEEE Conference on Games (CoG)*, pages 1–8, 2021.

[31] S. Gudmundsson, P. Eisen, E. Poromaa, A. Nodet, S. Purmonen, B. Kozakowski, R. Meurling, and L. Cao. Human-like playtesting with deep learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 08 2018.

[32] C. Guerrero-Romero, S. M. Lucas, and D. Perez-Liebana. Using a team of general ai algorithms to assist game design and testing. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, Aug 2018.

[33] D. Held, X. Geng, C. Florensa, and P. Abbeel. Automatic goal generation for reinforcement learning agents. *CoRR*, abs/1705.06366, 2017.

[34] J. Hernández Bécares, L. Costero, and P. Gómez-Martín. An approach to automated videogame beta testing. *Entertainment Computing*, 18, 08 2016.

[35] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[36] C. Holmgard, M. C. Green, A. Liapis, and J. Togelius. Automated playtesting with procedural personas with evolved heuristics. *IEEE Transactions on Games*, pages 1–1, 2018.

[37] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis. Evolving personas for player decision modeling. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[38] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis. Generative agents for player decision modeling in games. In *Proceedings of the 9th International Conference on the Foundations of Digital Games (FDG)*, 2014.

[39] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis. Monte-carlo tree search for persona based player modeling. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[40] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood. An automated model based testing approach for platform games. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 426–435, Sep. 2015.

[41] E. J. Jacobsen, R. Greve, and J. Togelius. Monte mario: Platforming with mcts. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 293–300, New York, NY, USA, 2014. ACM.

[42] S. James, G. Konidaris, and B. Rosman. An analysis of monte carlo tree search. In *AAAI*, 2017.

[43] M. Johnson, K. Hofmann, T. Hutton, D. Bignell, and K. Hofmann. The malmo platform for artificial intelligence experimentation. In *25th International Joint Conference on Artificial Intelligence (IJCAI-16)*. AAAI - Association for the Advancement of Artificial Intelligence, July 2016.

[44] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 341–348, Santorini, Greece, Sep 2016. IEEE. The best paper award.

[45] A. Khalifa, A. Isaksen, J. Togelius, and A. Nealen. Modifying mcts for human-like general video game playing. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 2514–2520. AAAI Press, 2016.

[46] J. G. Kormelink, M. M. Drugan, and M. Wiering. Exploration methods for connectionist q-learning in bomberman. In *ICAART*, 2018.

[47] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker. Combinatorial testing: Theory and practice. In *Advances in Computers*, volume 99, pages 1–66. Elsevier, 2015.

[48] L. Lee, B. Eysenbach, E. Parisotto, E. P. Xing, S. Levine, and R. Salakhutdinov. Efficient exploration via state marginal matching. *CoRR*, abs/1906.05274, 2019.

[49] C. Lewis, E. J. Whitehead, and N. Wardrip-Fruin. What went wrong: a taxonomy of video game bugs. In *FDG*, 2010.

[50] D. Lin, C.-P. Bezemer, Y. Zou, and A. E. Hassan. An empirical study of game reviews on the steam platform. *Empirical Software Engineering*, 24(1):170–207, 2019.

[51] D. Loubos. Automated testing in virtual worlds. Game and media technology msc, Utrecht University, 2018.

[52] T. Machado, D. Gopstein, A. Nealen, O. Nov, and J. Togelius. Ai-assisted game debugging with cicero. *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2018.

[53] B. Marklund, H. Engström, M. Hellkvist, and P. Backlund. What empirically based research tells us about game development. *The Computer Games Journal*, 8:1–20, 12 2019.

[54] B. Michini and J. P. How. Bayesian nonparametric inverse reinforcement learning. In P. A. Flach, T. De Bie, and N. Cristianini, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 148–163, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[55] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[56] L. Mugrai, F. Silva, C. Holmgård, and J. Togelius. Automated playtesting of matching tile games. In *2019 IEEE Conference on Games (CoG)*, pages 1–7. IEEE, 2019.

[57] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *CoRR*, abs/2003.04960, 2020.

[58] M. J. Nelson. Investigating vanilla mcts scaling on the gvg-ai game corpus. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–7, Sep. 2016.

[59] A. Y. Ng and S. J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[60] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[61] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis. Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93–104, 2013.

[62] I. Osband, C. Blundell, A. Pritzel, and B. V. Roy. Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4026–4034, 2016.

[63] G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos. Count-based exploration with neural density models. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2721–2730, 2017.

[64] M. Ostrowski and S. Aroudj. Automated regression testing within video game development. *GSTF Journal on Computing (JoC)*, 3(2):1–5, 2013.

[65] C. Paduraru, M. Paduraru, and A. Stefanescu. Rivergame - a game testing tool using artificial intelligence. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 422–432, 2022.

[66] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2778–2787, 2017.

[67] T. Pepels, M. H. M. Winands, and M. Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, Sep. 2014.

[68] D. Perez, S. Samothrakis, and S. Lucas. Knowledge-based fast evolutionary mcts for general video game playing. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, Aug 2014.

[69] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas. General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214, 2019.

[70] D. Perez Liebana, S. Mostaghim, S. Samothrakis, and S. Lucas. Multi-objective monte carlo tree search for real-time games. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 1–1, 01 2014.

[71] J. Pfau, J. Smeddinck, and R. Malaka. Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving. In *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play*, pages 153–164, 10 2017.

[72] C. Politowski, Y.-G. Guéhéneuc, and F. Petrillo. Towards Automated Video Game Testing: Still a Long Way to Go. *arXiv e-prints*, page arXiv:2202.12777, Feb. 2022.

[73] E. J. Powley, S. Colton, S. Gaudl, R. Saunders, and M. J. Nelson. Semi-automated level design via auto-playtesting for handheld casual game creation. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2016.

[74] E. J. Powley, P. I. Cowling, and D. Whitehouse. Information capture and reuse strategies in monte carlo tree search, with applications to games of hidden information. *Artif. Intell.*, 217(C):92–116, Dec. 2014.

[75] E. J. Powley, P. I. Cowling, and D. Whitehouse. Memory bounded monte carlo tree search. In *AIIDE*, pages 94–100. AAAI Press, 2017.

[76] C. Redavid and F. Adil. An overview of game testing techniques. *Västerås: sn*, 2011.

[77] N. Rhinehart and K. Kitani. First-person activity forecasting from video with online inverse reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2018.

[78] S. Roohi, A. Relas, J. Takatalo, H. Heiskanen, and P. Hämäläinen. *Predicting Game Difficulty and Churn Without Players*, page 585–593. CHI PLAY '20. Association for Computing Machinery, New York, NY, USA, 2020.

[79] A. Santos, P. A. Santos, and F. S. Melo. Monte carlo tree search experiments in hearthstone. pages 272–279, 08 2017.

[80] R. E. S. Santos, C. V. C. de Magalhães, L. F. Capretz, J. S. C. Neto, F. Q. B. da Silva, and A. Saher. Computer games are serious business and so is their quality: Particularities of software testing in game development from the perspective of practitioners. *CoRR*, abs/1812.05164, 2018.

[81] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J. B. Chaslot, and J. W. H. M. Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games*, pages 1–12, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[82] T. Schaul. An extensible description language for video games. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6:325–331, 12 2014.

[83] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[84] A. Sestini, A. Kuhnle, and A. D. Bagdanov. Demonstration-efficient inverse reinforcement learning in procedurally generated environments, 2020.

[85] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[86] D. Silver and G. Tesauro. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 945–952, 2009.

[87] D. J. N. J. Soemers, C. F. Sironi, T. Schuster, and M. H. M. Winands. Enhancements for real-time monte-carlo tree search in general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, Sep. 2016.

[88] A. Šošić, A. M. Zoubir, E. Rueckert, J. Peters, and H. Koeppl. Inverse reinforcement learning via nonparametric spatio-temporal subgoal modeling. *The Journal of Machine Learning Research*, 19(1):2777–2821, 2018.

[89] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[90] M. J. W. Tak, M. H. M. Winands, and Y. Bjornsson. N-grams and the last-good-reply policy applied in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, June 2012.

[91] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel. #exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 2753–2762, 2017.

[92] B. Tastan and G. Sukthankar. Learning policies for first person shooter games using inverse reinforcement learning. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'11, pages 85–90. AAAI Press, 2011.

[93] A. Tychsen and A. Canossa. Defining personas in games using metrics. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, pages 73–80, New York, NY, USA, 2008. ACM.

[94] N. Tziortziotis, K. Tziortziotis, and K. Blekas. Play ms. pac-man using an advanced reinforcement learning agent. In A. Likas, K. Blekas, and D. Kalles, editors, *Artificial Intelligence: Methods and Applications*, pages 71–83, Cham, 2014. Springer International Publishing.

[95] S. Varvaressos, K. Lavoie, S. Gaboury, and S. Hallé. Automated bug finding in video games: A case study for runtime monitoring. *Comput. Entertain.*, 15(1):1:1–1:28, Mar. 2017.

[96] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[97] M. Wulfmeier, P. Ondruska, and I. Posner. Maximum entropy deep inverse reinforcement learning. *arXiv preprint arXiv:1507.04888*, 2015.

[98] C. Xiao, J. Mei, and M. Müller. Memory-augmented monte carlo tree search. In *AAAI*, pages 1455–1462. AAAI Press, 2018.

[99] D. Ye, G. Chen, W. Zhang, S. Chen, B. Yuan, B. Liu, J. Chen, Z. Liu, F. Qiu, H. Yu, Y. Yin, B. Shi, L. Wang, T. Shi, Q. Fu, W. Yang, L. Huang, and W. Liu. Towards playing full MOBA games with deep reinforcement learning. *CoRR*, abs/2011.12692, 2020.

[100] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 772–784, 2019.

[101] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *Proc. AAAI*, pages 1433–1438, 2008.

[102] E. İlhan and A. Ş. Etaner-Uyar. Monte carlo tree search with temporal-difference learning for general video game playing. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 317–324, Aug 2017.

[103] M. Świechowski, J. Mańdziuk, and Y. S. Ong. Specialization of a uct-based general game playing program to single-player games. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):218–228, Sep. 2016.

APPENDIX A

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Arıyürek, Sinan
Nationality: TC
Date and Place of Birth:
Marital Status: Married

EDUCATION

Degree	Institution	Year of Graduation
Ph.D.	Information Systems, METU	2022
M.Sc.	Computer Engineering, Bilkent	2012
B.Sc	Computer Engineering, Bilkent	2009

PROFESSIONAL EXPERIENCE

Company	Position	Year
Luxoft/ASML	Senior Software Developer	2019-Present
Simtek	Software Developer	2014-2018
Freelancer	Software Developer	2012-2013

PUBLICATIONS

- S. Ariyurek, A. Betin-Can and E. Surer, "Automated Video Game Testing Using Synthetic and Humanlike Agents," in *IEEE Transactions on Games*, vol. 13, no. 1, pp. 50-67, March 2021, doi: 10.1109/TG.2019.2947597.
- S. Ariyurek, A. Betin-Can and E. Surer, "Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing," 2020 *IEEE Conference on Games (CoG)*, 2020, pp. 25-32, doi: 10.1109/CoG47356.2020.9231670.
- S. Ariyurek, E. Surer and A. Betin-Can, "Playtesting: What is Beyond Personas," in *IEEE Transactions on Games*, doi: 10.1109/TG.2022.3165882.