

Automated Video Game Testing Using Synthetic and Human-Like Agents

Sinan Ariyurek, Aysu Betin-Can, Elif Surer

Graduate School of Informatics

Middle East Technical University

06800, Ankara, Turkey

{sinan.ariyurek, betincan, elifs}@metu.edu.tr

Abstract—In this paper, we present a new methodology that employs tester agents to automate video game testing. We introduce two types of agents—synthetic and human-like—and two distinct approaches to create them. Our agents are derived from Reinforcement Learning (RL) and Monte Carlo Tree Search (MCTS) agents, but focus on finding defects. The synthetic agent uses test goals generated from game scenarios, and these goals are further modified to examine the effects of unintended game transitions. The human-like agent uses test goals extracted by our proposed multiple greedy-policy inverse reinforcement learning (MGP-IRL) algorithm from tester trajectories. MGP-IRL captures multiple policies executed by human testers. These testers’ aims are finding defects while interacting with the game to break it, which is considerably different from game playing. We present interaction states to model such interactions. We use our agents to produce test sequences, run the game with these sequences, and check the game for each run with an automated test oracle. We analyze the proposed method in two parts: we compare the success of human-like and synthetic agents in bug finding, and we evaluate the similarity between human-like agents and human testers. We collected 427 trajectories from human testers using the General Video Game Artificial Intelligence (GVG-AI) framework and created three games with 12 levels that contain 45 bugs. Our experiments reveal that human-like and synthetic agents compete with human testers’ bug finding performances. Moreover, we show that MGP-IRL increases the human-likeness of agents while improving the bug finding performance.

Index Terms—Reinforcement Learning, Monte Carlo Tree Search, Automated Game Testing, Inverse Reinforcement Learning, Graph Coverage.

I. INTRODUCTION

Video games industry is a multi-billion industry that is continually growing [1]. Though the success of a video game depends upon numerous aspects, the presence of bothersome bugs decreases the overall experience of a player. Moreover, the bugs that are found after release not only increase the overall budget [2], but also act as a negative feedback on the development and testing team. Hence, the game is tested painstakingly by game developers and players, and these tests require immense tester effort. The major difficulty of game testing arises from the constant changes [3]. Therefore, researchers proposed various methods to decrease the test effort. These methods are regression tests based on record/replay segments [4], scenario testing [5], UML-based sequence generation [6], agents harnessing Petri nets [7] and RL agents [8][9] to automate this process. These studies, however, lack

either human expertise, an automated oracle, an intelligent tester agent, or an overall game testing experiment.

In game playing, researchers employed artificial intelligence (AI) to make agents behave like human beings from collected human data, and called these agents as human-like agents in the literature. Human-like agents are better suited to analyze the difficulty of the game [10], can become genuine opponents [11][12], and can generate satisfying playthroughs. In game testing, these collected human data are used to perform regression testing rather than creating an intelligent agent. During alpha and beta testing phases [13], countless test data can be collected from players. Human game testers participating in these phases use their expertise to examine the game. We propose a method to capture this expertise in the form of test goals and use these test goals in agents so that they can test like the original human player. Test goals are objectives that agents want to validate in a game. Test goals range from whether the game can be finished to whether the agent can walk through walls. Depending on the test goal, agents generate a different test sequence. In this paper, the goals that are extracted from collected human data are called human-like test goals.

On the other hand, a game can be viewed as the implementation of the game designer’s story. This story—whether linear or non-linear—can be represented using a graph [14]. In this paper, this graph is referred to as a game scenario graph. Game scenario graph (see Fig. 1) is designed by the game designer and contains high-level behavior. A node on this graph is realized with the states of the game (see Appendix A Fig. 9). Edges are the actions that progress the story. Additionally, as directed graphs form the foundation of several coverage criteria in software testing [15], it is possible to generate test sequences as paths using this graph and a coverage criterion. We enhance the test sequence by adding actions at each node that should not progress the game. The criterion-based paths verify the implementation of the game scenario while the enhancements check other aspects of the game such as testing collisions and unintended actions. We propose a method that translates these ideas into test goals. Since no human data are used, we call these goals synthetic test goals.

Game researchers used RL agents to play various games such as Ms. Pac-Man [16], Bomberman [17], Unreal Tournament [11][12]. Furthermore, recent developments in AI showed that agents can surpass humans in arcade games [18], Go [19] and StarCraft II [20]. The success in Go

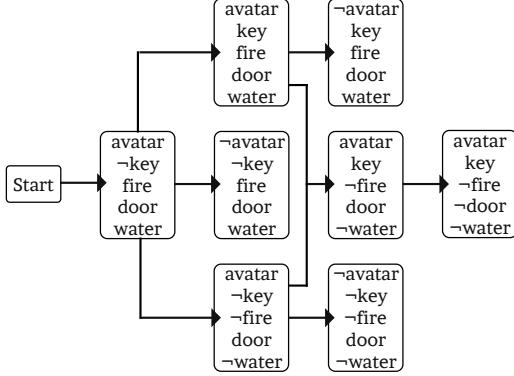


Fig. 1: A scenario graph model using atomic properties

[19] is achieved with RL, Supervised Learning, and MCTS. Moreover, MCTS [21] agents are found to be successful on GVG-AI [22] and General Game Playing (GGP) [23], which are the most well-known frameworks that explore agents that can play various games. In our system, MCTS and RL agents use the synthetic or human-like test goals to generate test sequences. In this paper, a synthetic agent is an RL or an MCTS agent which uses synthetic test goals, and a human-like agent is an RL or an MCTS agent which uses the extracted human test goals.

Current off-the-shelf automated testing tools help testers to implement and automate test scenario executions. However, these tools do not design test scenarios or test sequences. Our study fills this gap with intelligent agents. Our synthetic and human-like agents generate test sequences for a game under test without human intervention. Then, our framework replays these generated test sequences and check the game behavior with an automated oracle. The test oracle checks if the game behaves as expected according to the game design constraints and game scenario graph which are given by the game developer. Our aim is to find discrepancies between the actual implementation and the design of the game, thus visual glitches are not checked by our oracle. We present our approach using the GVG-AI framework. GVG-AI is chosen since it contains numerous games, and has a data collection mechanism. In this paper, our approach is demonstrated via grid games, but it can be generalized to different game models.

Our first contribution is human-like tester agents, to the best of our knowledge, our work is the first to propose these agents in game testing. The proposed human-like agents can be used to test other levels, thus possess an advantage over record/replay testing. Second, the synthetic agent is an improvement over simple scenario testing, rewarding the examination of all allowed transitions and some unallowed game transitions for robustness. Lastly, we present an interaction state that enables us to capture tester strategies and play them.

This paper is structured as follows: Section II gives preliminary information about Graph Testing, RL, MCTS and GVG-AI. Section III describes the examples and methodologies of related research. Our proposed methodology is presented in Section IV while the details of our experiments are illustrated

in Section V. Section VI discusses the outcomes of the strategies used, their contributions and limitations. Section VII concludes this paper.

II. PRELIMINARIES

The following subsections introduce the preliminary material: Graph Testing, RL, MCTS, and GVG-AI.

A. Graph Testing

In software testing, there are several systematic ways to create tests and evaluate the adequacy of the test set. Testers model the system under test (SUT) and check for defects with the help of this model. One common way to model software is using graphs. Once we model the SUT with a graph, we can apply systematic testing techniques to generate test scenarios.

A directed graph is a structure that consists of nodes N and edges E where $E \subseteq N \times N$. A path is a sequence $[n_1, n_2, \dots, n_M]$ of nodes where each pair is in the set of edges E . For graphs, various sequences can be obtained depending on the coverage criterion. The coverage definitions are stated from basic to complex. **Edge Coverage (EC)**: Contains each reachable path of length up to one. **Edge-Pair Coverage (EPC)**: Contains each reachable path of length up to two. **Simple Path (SP)**: Requires a node not to appear more than once, unless it is the initial or final node of the path. **Prime Path (PP)**: Strengthens the simple path definition by requiring the path not to be a sub-path of another simple path. **Prime Path Coverage (PPC)**: Contains each prime path in the graph. **All Path Coverage (APC)**: Contains every path in the graph. The order of coverage is: $EC \subseteq EPC \subseteq PPC \subseteq APC$ [15].

B. Reinforcement Learning

In reinforcement learning (RL), an agent experiences an environment, examines its current state and interacts with it. The agent receives a reward and a new state from the environment as a response to these interactions. Markov Decision Process (MDP) defines this experience between an agent and the environment [24], and the RL problem can be written using an MDP. Markov Decision Process is a tuple (S, A, T, R, γ) where S is the set of states, A is the set of actions, and $T : S \times A \times S \rightarrow [0, 1]$ is the transition probability. The reward function is $R : S \times A \rightarrow \mathbb{Q}$, and γ is the discount rate for future rewards.

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

In RL, the goal is to find the action to take given a state, which is the rough definition of the policy. Formally, a policy is a probability distribution that maps actions over given states. State-action-reward-state-action (Sarsa) is an on-policy model-free temporal difference learning algorithm which is shown in (1). $Q(s, a)$ defines the Q-function which represents the expected total reward of taking action a in state s . $R(s, a)$ represents the immediate reward of taking action $a \in A$ in state $s \in S$, α is the learning rate, and δ is the temporal difference error. For each episode, this equation is iterated starting from

an initial state until a certain criterion is met. On-policy methods update the current estimate by the action selected from the policy. $Q(s, a)$ is updated using the $Q(s', a')$ where $a' \in A$ is selected from the current policy. On the other hand, off-policy methods such as Q-Learning update the current estimate by the action selected from another policy. Lastly, model-free methods do not require knowing the dynamics (T) of an environment.

The temporal difference learning is a *bootstrapping* method where the previous estimate is used to update the new estimate. In Sarsa, only the current $Q(s, a)$ is updated, but Sarsa(λ) uses eligibility traces. In eligibility traces, every state visited during an episode is marked as eligible for update, and each iteration also updates the states that are marked as eligible. The eligibility of a state is decayed by λ and γ .

The Q-function can be represented using a table of state and action pairs (tabular), or by a function approximator such as a neural network. Last but not least, there is the dilemma of exploration/exploitation. Exploration is gathering more knowledge and exploitation is choosing the best action with the current knowledge. The agent's objective is to maximize the total expected reward; therefore, it has to balance exploration/exploitation.

C. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [21] is a search method that iteratively expands the search tree in the preferred direction. MCTS executes four consecutive steps iteratively until a certain condition occurs. This condition can be—including but not limited to—reaching the desired terminal node or expiration of the allowed computational budget. These four steps are selection, expansion, simulation, and backpropagation. The selection phase selects a node from the tree according to a Tree Policy. An acclaimed approach is to use the UCB1 algorithm.

This approach balances the exploration and exploitation of the search. In the expansion phase, one of the unexplored children of the selected node is added to the search tree. The simulation phase starts with this node, and a default policy is used to sample moves. The score obtained at the end of this simulation is used to update the values of the nodes, starting from the simulated node propagating up to the root node. This is the backpropagation phase. These four phases are repeated in this order until the computational budget expires. Afterwards, depending on a criterion, a child of the root node is returned.

D. GVG-AI

GVG-AI [22] is a framework that contains single/multiplayer 2D games. There are more than 120 games just for the single player, which include well-known games such as Mario, Zelda, and Sokoban. Due to the variety of games, GVG-AI poses a challenging and interesting environment. GVG-AI games also hold another special property: they are all defined using a language called VGDL [25]. This language defines the game rules in a specific game level, such as what will happen if the avatar attacks an enemy, or interacts with a key. We

slightly modified the GVG-AI to access all of the interactions amongst all sprites including the hidden sprites.

In this study, we consider the bugs between the game implementation and the game design. GVG-AI framework creates a game by transforming the game source code written in VGDL. Note that, we are not testing the internal engine that makes this transformation, but the created game. Thus, VGDL is the implementation, and the game scenario graph along with the game constraints are the game design.

III. RELATED RESEARCH

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 [3]. To validate these attributes, researchers proposed various methods to generate test sequences: record/replay [4], handcrafted scenarios [5], UML and state machines [6], Petri nets [7], and RL [8][9]. However, when game environment changes, test sequences and scenarios such as [4] [5] become obsolete and a manual tester effort is required to create a new test sequence. As opposed to these manual techniques, UML based techniques automate the process of test generation. However, generating sequences from UML runs into state explosion for larger games; and without a gameplay AI, it relies on the generated states to play the game. Hence, researchers employed AI to test games. In [7], an AI played according to a petri net description of the game that contains high level actions, but they only generated test sequences that cover the game scenario. RL used in [8] with short and long term memory is useful, but the approach is limited to the Point-and-Click games. The agent in [9] can roam, but only the crafting system is tested using RL. More generalized approaches such as [26] experimented on two GVG-AI games for finding bugs. As the authors of [26] stated, since the agent was a gameplaying agent, the agent was not required to find all of the bugs. Lastly, [27] introduced a team of agents with different purposes to test games. Nonetheless, all of these studies lack controlled experimentation.

Game Playing: Researchers applied RL and MCTS to numerous games, and there are plentiful studies on these topics. For example, Sarsa(λ) is used as a game playing agent in Ms. Pac Man [16] and to create a human-like agent in Unreal Tournament [12]. Although the aim of these papers was 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. 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 [28] [29]. Amongst them, knowledge-based evaluation (KBE) [30] is found beneficial. Therefore, we used KBE enhancement in our MCTS.

Learning From Humans: Incorporating domain knowledge is prominent, both in RL and MCTS, yet defining the correct set of rewards is a hurdle. Moreover, even humans learn better when guided by or imitating an expert. Inverse

reinforcement learning (IRL) is the study of extracting a reward function, given an environment and observed behavior, which is sampled from an optimal policy [31]. IRL methods exist in order to extract reward function when i) trajectories that are sampled from the same policy [32] [33] [34], ii) trajectories that are sampled from different policies [35], and iii) trajectories that are better explained with multiple sub-goals [36] [37], are given. The trajectories that were collected from testers fit into the last category since the testers can test several goals in the same run. However, as noted by [37], [36] fails to generalize to unseen states and the approach in [37] finds sub-goals in the same level. Therefore, we propose MGP-IRL overcome these drawbacks. Additionally, human-like agents are used in games such as Unreal Tournament [11], Super Mario [38], Catan [39], GVG-AI [40], Spades [41], and Candy Crush [10]. However, we focus on human-like tester agents.

IV. METHODOLOGY

We propose an agent based system to automate game testing to detect the discrepancies between game design and game execution. To this purpose, two types of test goals are created: a) synthetic test goals based on the game scenario, rewarding all valid and some invalid transitions, and b) human-like test goals that are trained from human tester data. Using these test goals in RL and MCTS agents, we generate test sequences and check if the game software behaves according to the game constraints automatically.

In this study, the term “interaction” does not refer to the interaction definition used in GVG-AI framework, but it refers to our definition of interaction, which is proposed for game testing purposes.

In the following sections, first, we define the interaction state. Then, we present our approach for synthetic test goal creation. We continue by explaining our method in learning test goals from the human data. Finally, we describe how to generate test sequences, and our proposed oracle.

A. Interaction State

Testers exercise a game by following several strategies [13] which lead them to interact with various aspects of the game. Testers mark these interactions as “tested” in a memory. This memory prevents testers from executing the same interactions. Memory can be a pen and paper, a tool or the intangible memory of the testers.

Interactions that progress the game, such as picking up a key can be modeled using an MDP. On the other hand, it is difficult to model the interactions that do not advance the game, such as trying to pass through a wall. If the game does not allow this behavior, the player should stay in the same position. Moreover, if a positive reward is obtained with this interaction, an agent can repeat the hitting the wall interaction infinitely. A memory can solve this state representation by marking this wall as “tested”. The game state does not record this interaction, but the memory does. Therefore, the MDP formulation becomes simpler when we use both game state and interaction state together.

An idea of memory is used by Pfau et al. [8] to explore the states of a Point-and-Click adventure game. They used this memory idea to adjust the reward of each available action. However, we use memory to record the interactions performed during testing. In this study, we examine automated testing approaches on 2D grid games. Therefore, we use a grid-based memory model. This grid is referred to as the interaction state, as it marks interactions. The interaction state is a supplementary state to the game state. The game state is a grid that holds the positions of sprites in the game. Using only the game state to model testing behavior by MDP is inadequate for the following reasons. First, only specified VGDL game rules can alter the game state; thus, some of the interactions will not alter the game state at all. Second, due to a bug, even state changing interactions may not manifest on the game state.

In 2D grid games, interactions occur between sprites. Therefore, we propose an interaction as a tuple: $\zeta = \langle \eta_0, \eta_1, Pos, Dir, Type, AvatarState \rangle$ where $\eta_0, \eta_1 \in \Gamma$ and Γ is the set of sprites, $Pos : \langle x, y \rangle$ is the position where the interaction took place, $Dir \in \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ is the direction of the interaction calculated from the first sprite’s direction (η_0), $Type \in \{Move, Use\}$ where *Use* can be mapped to actions such as *Attack*, $AvatarState$ represents the states that an avatar can take —for VGDL they are listed under avatar definition. We should note that, the first three parameters are mandatory while the rest are optional.

We use a grid-based model to represent the interaction state since we focus on grid games. An interaction state is modeled using a 3D grid to identify the following concerns: the tester may prioritize testing a sprite from all directions, the tester may exercise an interaction more than once, and the tester may choose to differentiate between a movement and *Use*. A 3D grid solves these concerns while keeping a single number in every grid cell. This number counts how many times an interaction has occurred. Every layer in this 3D grid is a 2D grid which has the width and height of the game grid. In the final implementation, we used 12 2D grids. First four layers represent the movement interactions done between the avatar and other sprites. Each of these four layers represents a different direction. Next four layers represent interactions with *Use*, again one for each direction. The last four layers represent the interactions that do not fit the aforementioned levels; such as when the avatar pushes the water bucket and the water bucket also moves or when an enemy interacts with another sprite. Once more, it holds a layer for each direction.

Although we proposed an interaction state for grid games, it can be applied to other games as well. The basic idea is to duplicate the visual environment for the purpose of marking tested positions, and this duplication process can be automated. Furthermore, additional layers can be added depending on what needs to be differentiated between different categories: movement, use, and so on. If the objects such as keys or doors have more importance, rather than creating a replica of the environment, an array containing just these preferred objects can be created.

For the rest of this paper, we use the following definitions. A Game is a tuple $\langle \mathcal{G}, \mathcal{I}, A, \Gamma, \delta, ||\zeta|| \rangle$, where \mathcal{G} is the set of game states, \mathcal{I} is the set of interaction states, $A = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$

```

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/wall13

InteractionSet
  movable wall > stepBack
  nokey goal   > stepBack
  goal withkey > killSprite
  nokey key    > transformTo stype=withkey
                  killSecond=True

```

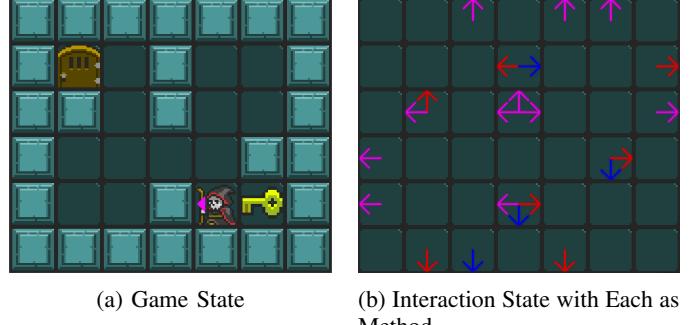
Fig. 2: Simplified SpriteSet and InteractionSet in VGDL

, *Use*, *Nil*} is the set of all actions where the availability of *Use* depends on the game, and *Nil* signifies taking no action. Γ is the set of sprites and δ is a transition function that takes an action and outputs the next game state, interaction state, the interactions, and $||\zeta||$ is the set of all interactions. S in MDP (see Section II-B) is defined as $S : \mathcal{G} \times \mathcal{I}$.

Lastly, a feature is a tuple $\phi = \langle \eta_0, \eta_1, Weight, Method, Type, Rep, AvatarState \rangle$ where $\eta \in \Gamma$, *Weight* is the reward obtained from this feature. *Method* represents the direction preference of the tester. We have two different proposals for this parameter {*Each*, *All*}. *Each* is used to differentiate different directions whereas *All* is used for considering all of the directions to be the same. *Type* is {Move, *Use*}. *Rep* limits the number of times that the reward can be obtained. Lastly, *AvatarState* is the same as that of interaction.

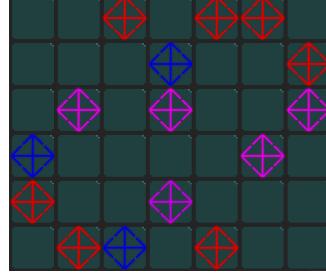
An action taken on a game state generates interactions, and interactions are matched with features which are used to calculate the reward. When an interaction i occurs, the feature ϕ_i corresponding to this interaction is retrieved from the feature set of the game using $\eta_0, \eta_1, Method, AvatarState$ values of the interaction. Then, the *Rep* of ϕ_i is compared with the number of times the interaction i has occurred, which is stored in the interaction state. If this value is less than or equal to the *Rep*, the reward defined in *Weight* is acquired and the value in the cell is incremented. *Each* only updates a single cell, whereas *All* updates all of the cells. This is how the *reward function* is calculated. Lastly, if an interaction does not match with a feature, the interaction state will stay the same.

To illustrate the game and the interaction states, consider the game transcribed in VGDL in Fig. 2. We illustrate the game state and two different interaction states in Fig. 3. A game state that contains the sprites $\Gamma = \{Avatar, Door, Floor, Key, Wall\}$ is shown in Fig. 3(a). The magenta triangle on the avatar represents its direction, and $AvatarState = NoKey$. In Fig. 3(b) and Fig. 3(c), two interaction states are shown, where interactions in the first four layers are represented with red color while interactions in layers four to eight are represented with blue color. Their combination is depicted in magenta. The arrows represent the direction of the interaction saved in the state. In Fig. 3(b), since individual arrows are shown, it can be understood that the features' *Method* parameter is *Each*, whereas in Fig. 3(c)

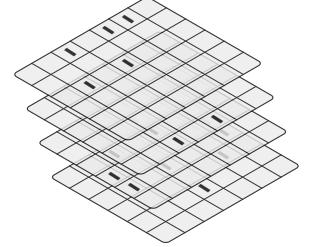


(a) Game State

(b) Interaction State with Each as Method



(c) Interaction State with All as Method



(d) Interaction State Layers from Fig. 3(b)

Fig. 3: Game State and Interaction State

individual arrows are not shown, thus the features' *Method* parameter is *All*. In Fig. 3(d) the first four layers of Fig. 3(b) are dissected, and for convenience cells that have zeros are left empty. Recall that, the numbers in cells show how many times an interaction is performed.

In this grid, the top-left corner is $<0, 0>$. The avatar in Fig. 3(a) is at position $<4, 4>$. For example, in Fig. 3(b), the direction of the interaction at position $<2, 0>$ is \uparrow . There could be more than one interaction at a position. For example, in Fig. 3(b) at position $<3, 4>$ all four directions except \uparrow are exercised at this cell. In Fig. 3(b), we see that the agent executed $<Avatar, Wall, <2, 0>, Move, \uparrow, NoKey>$ and the feature was $<Avatar, Wall, 1, Move, Each, 1, NoKey>$ since there is only an upward arrow. In Fig. 3(c), the following features are used $<Avatar, Wall, 1, Move, All, 1, NoKey>$ (see the blue and magenta marked cells), $<Avatar, Wall, 1, Use, All, 1, NoKey>$, (see the red and magenta marked cells). This interaction state shows that, in the top middle walls, the tester executed the *Use* and *Move* interactions.

B. Test Goal Generation

1) *Synthetic Test Goals*: Automation of the testing steps is crucial as it speeds up development while reducing the testing effort. Although approaches like playing a pre-defined or pre-captured scenario are effective, they are no longer usable when the game layout changes. In this study, we present a game scenario graph-based approach to create synthetic test goals.

We use the game scenario graph to generate various scenario paths using a graph coverage criterion. In our case, the game scenario graph defines only allowed transitions. These generated paths target different game routes in the scenario.

Our aim is to create test goals from these paths that an agent understands. In this regard, we use features to guide the MCTS or RL agent towards these generated test goals. Hence, our agents can verify if the game scenario is implemented correctly in a game.

In addition to checking if the game implements the scenario, we also want to check whether the game implements some additional behavior that is **not** in the scenario, as any tester would do. This promotes the agent to ask questions like: What happens if I attack the key? Can I pass through the walls? We generate a list using all combinations of the following four parameters of a feature $\langle \eta_0, \eta_1, Type, AvatarState \rangle$ and this list is referred to as modifications list. For games with high sprite count, pair-wise combinatorial [42] strategy can be used in the generation of the modification list. We initially prune the modification list by restricting η_0 to be movable.

Synthetic test goal generation algorithm is as follows: The developer gives a game scenario graph, such as Fig. 1, and a table that maps the edges to abstract features. A feature is abstract when *Weight*, *Method*, and *Rep* parameters are empty. Given this scenario graph and a coverage criterion, the system first generates paths that need to be covered during test execution. Then, each path of the form $[n_1, n_2, \dots, n_M]$ is converted into a sequence of features by replacing each pair of nodes (n_i, n_j) , i.e. edge, with the abstract feature retrieved from the given table. After this conversion, the paths are discarded, and the process continues with the feature sequences. Next, each of this generated feature sequence is modified by inserting an extra feature from the modification list. This extra feature can be inserted to any place in the sequence. When modifying a sequence of features, the Manhattan distance between the candidate feature from the modification list and every feature in the sequence should be at least one. Manhattan distance is calculated by comparing the equality of $\eta_0, \eta_1, Type, AvatarState$ parameters of features. If a compared parameter is the same, its distance is zero and one if they are not the same. This control prioritizes the features that have not been included in the original feature sequence.

For a feature sequence that has M features and K number of modifications, if we choose to insert all modifications to the sequence at the same time, we can insert up to $M \times K$ modifications —not considering the order of modifications. Nevertheless, this approach is inadequate for testing, a bug found while agents using a feature from modification list might crash the game and the bugs will not be noticed which can be found by using other features from this list. Therefore, to prevent this “masking” and to reduce the complexity of the feature sequence to be played, we copy the original feature sequence $M \times K$ times. Then insert one modification into each copy. We also include the original feature sequence that is not modified. Therefore, from a single feature sequence, $M \times K + 1$ number of feature sequences are generated at most. It should be noted that an abstract feature is concretized by setting the other three feature parameters manually. In this paper, *Rep* parameter is set depending on η_1 : 3 if it is a movable sprite, 1 if it exists in abundance such as walls, else set as 2. *Method* is set as *All*, and the parameter *Weight* is set as one.

This approach generates features that will guide the agent through different parts of the game. For a more complex graph, these features can become difficult to play. Since every edge in a game scenario path corresponds to a feature, the number of features scale with the number of edges in this path. Hence, in this study, we divide the overall path, influenced by the sequential approach of Rhinehart and Kitani [43]. This division is crucial due to two factors: first, we want the agent to execute the feature sequence in the intended order so that the agent traverses the scenario graph in the intended order; second, due to a bug the overall path may not be played, but separation helps pinpoint exactly in which part the problem occurred.

We present a different goal state definition from [43][36][37]. Our goal state —test goal— is defined as S_h where $S_h \subseteq S$. In the case of a goal such as testing all walls or covering all empty spaces, there are many states in S_h that represent this goal. Moreover, we should be able to define a goal where the agent tests a single wall. However, this definition does not allow a flexible mechanism to specify this goal. To this purpose, we propose a criterion for each feature. A criterion specifies the percentage of η_1 to be tested, and the combination of features and criteria constitutes a goal. Consequently, while features guide the agent through the grid, the criteria check whether the agent fulfilled the goal.

When the goal is fulfilled, i.e. the agent is in S_h , the agent moves on to the next goal in the sequence. The agent may not fulfill a goal due to two reasons. First, the goal may be infeasible, such as attacking a door which is hidden. Second, a bug may prevent reaching a goal. Hence, we may choose to terminate or move on to the next goal if an agent does not reach a goal. In our implementation, we made this choice optional. Furthermore, to assist exploration, the feature referred to as the exploration feature $\langle Avatar, Floor, All, Move, 0.01, 1, AvatarState \rangle$ is added to every goal, and *AvatarState* parameter is copied from the other feature in the same goal. In synthetic test goals, we set criterion values to 100% for the features from the original sequence and 0% for the exploration feature. The agent acquires a positive reward for traveling, but exploration is not required for the agent to pass the criteria. Formally, the goal h consists of features and a criterion for each feature $\{(\phi_0, c_0), (\phi_1, c_1), \dots, (\phi_n, c_n)\}$, where c_0, \dots, c_n are positive real numbers. Hence, the synthetic test goal approach generates a sequence of goals $\mathcal{H} = (h_0, \dots, h_n)$.

2) *Human-Like Test Goals*: Beta-testing is an invaluable part of the game development process. Human testers participating in beta-testing use their expertise and heuristics to uncover various bugs. During any test phase, the actions of each participant can be recorded as trajectories. A trajectory τ is a sequence of actions $[a_0, \dots, a_n]$ where $a_i \in A$ and $0 \leq i \leq n$. A trajectory itself does not represent anything meaningful, but, when the trajectory is replayed, it exhibits the intentions of a tester. Therefore, in the literature, the collected trajectories are used in regression testing [4]. In this study, we aim to capture the human testers’ expertise and automate the test generation by learning from the actions of these testers instead of repeating/replaying them. In this regard, inverse

reinforcement learning (IRL) is chosen to grasp this expertise. IRL assumes that the trajectory is near-optimal, but during ad hoc testing this assumption may not hold. Moreover, the human tester may perform a complex sequence of actions that cannot be modeled by linear weights [36][37]. Therefore, we propose to automatically partition these trajectories so that these partitions are near-optimal, as described below.

We propose MGP-IRL to capture tester expertise, which is presented in Algorithm 1. First, at line 2, the algorithm replays the actions in the trajectory and splits them to minimal trajectories and interactions. We split the trajectory at points where interactions change. We define change as any variation in $\eta_0, \eta_1, Type$ parameters of an interaction. At line 3, the set of previous features Φ_a is initialized as empty set, sequence of previous trajectory τ_a is set as empty sequence, the previous likelihood threshold κ_a is initialized as zero, and the goal sequence \mathcal{H} is set as empty sequence.

In lines 6 to 8, the current segment is converted to a feature ϕ_i and Φ_b becomes the union of the feature ϕ_i and the previous features Φ_a . This feature discovery allows the algorithm to employ non-zero weight strategy for unobserved interactions [32]. This strategy shrinks the feature space and supports learning with fewer expert trajectories. Note that parameters such as *Weight*, *Method*, *Rep* cannot be captured from interactions, these parameters are left empty at this step. At line 9, the trajectory τ_b is replayed with the features Φ_b to find the repetition count of Φ_b . Without this step, our IRL procedure will be ill-posed since there is a reward that the agent can acquire for unknown amount of times.

At line 10, IRL is applied to the trajectory τ_b to find the weights of the features Φ_b . Next, at line 11, the likelihood [35] of the trajectory κ_b is calculated using Φ_b . This likelihood estimate is used to decide whether we should combine trajectories and features. We know that trajectory τ_a can be sampled from a policy π_a that uses Φ_a with the likelihood of κ_a , and question whether τ_i should be added to τ_a which creates a new policy π_b that uses Φ_b with the likelihood of κ_b . If the combination of these trajectories is more likely, the difference of $\kappa_a - \kappa_b$ is expected to be negative.

If the condition at line 12 holds, the previous features, likelihood, and trajectories are replaced with the current features, likelihood, and trajectories. If we cannot combine the trajectories, at line 16, the previous segment is converted into a goal by calculating the criterion of each feature in Φ_a —if the feature holds a non-negative reward.

During experiments, we examine the effect of the threshold κ_T . The lower the threshold, the more likely the agent can repeat these interactions. As the threshold increases, the algorithm behaves more similarly to the internal IRL algorithm.

To calculate the criteria, first, the previous trajectory τ_a is replayed to count how many times each feature in Φ_a is seen in interaction state. This count is represented as $countF(\phi_i)$ where $\phi_i \in \Phi_a$. Then, for each ϕ_i , we count the number of occurrences of η_1 in the game state. This count is represented as $countS(\eta_1)$. For example, in the game state shown in Fig. 3(a), $countS(Wall)=27$ and $countS(Door)=1$. Finally, the criterion for a feature ϕ_i is calculated by $\frac{countF(\phi_i)}{countS(\eta_1)} \times 100$. This division is used to normalize the $Count(\phi_i)$ and this

normalization supports achieving a similar behavior in other levels. Since the algorithm is finished with this segment, it progresses the game state by applying each action in the trajectory τ_a . Note that the interaction state in the game state is reset after this step. This procedure continues until every segment is processed.

At line 19, the algorithm checks if there is a remaining segment, and if there is a segment, it is converted into a goal. Lastly, as stated in the interaction state, not every tester will interact with the same feature in the same way. Therefore, lines from 6 to 11 (not shown in the algorithm) are performed twice by changing the *Method* parameter of ϕ_i to obtain whether the tester has a direction preference. We chose the one that yields a higher likelihood.

To sum up, by splitting the trajectory using the interactions, we are able to use non-zero weights, find an estimate for the repetition of the features, calculate the direction preference of testers, and most importantly split the trajectory into policies that fall under a certain likelihood threshold. Consider the game depicted in Fig. 3(a) and consider that a human executed the following trajectory. First, she tried to go through the door, but on the way attacked the walls, and then attacked the key. MGP-IRL dissects this trajectory into interactions described as above. In the first iteration, the weights, method, repetition limit for walls are found. In the second iteration, the algorithm considers merging the first trajectory with the trajectory that includes the door. Since the tester moved towards the goal and attacked the walls along this way, the likelihood of the combined trajectory increases. Therefore, the algorithm combines these trajectories and sets the current trajectory as the combined trajectory. In the third iteration, the algorithm considers merging it with the trajectory that includes the Key. This combination will decrease the likelihood of the trajectory as the tester can execute this action early in the trajectory. Depending on the selected κ_T , the threshold check may fail. Then, the initial combined trajectory is converted into a goal, and the remaining trajectory is converted into another goal. Finally, these two goals are inserted into the goal sequence \mathcal{H} consecutively. If the threshold check passes, the whole trajectory is converted into a single goal.

Our approach is different from [36] and [37], since their sub-goal definition is based on the state, which restricts the applicability of the extracted test goals to other levels. Moreover, due to our feature definition, we should know the number of repetition and direction preferences, which are not computed in their approach.

C. Generating Test Sequences

Testing is a rigorous agenda, and an update to the code or design requires re-testing. Thus, the testing cycle becomes more tiresome over time. Therefore, the ability to automatically create new sequences is necessary. To fulfill this necessity, we propose using agents to create new test sequences automatically using test goals. This section explains how our agents generate new test sequences.

Learning algorithms take an environment and a reward vector, but our agents contain a goal sequence. Therefore, an agent

Algorithm 1 Multiple Greedy Policy Inverse Reinforcement Learning for extracting test goals from human trajectories

```

1: procedure MGP-IRL( $G, \tau, \kappa_T$ )
2:    $\zeta_{0..n}, \tau_{0..n}, n \leftarrow \text{SPLITTRAJECTORY}(G, \tau)$ 
3:    $\Phi_a \leftarrow \{\}, \tau_a \leftarrow [], \kappa_a \leftarrow 0, \mathcal{H} \leftarrow []$ 
4:    $i \leftarrow 0$ 
5:   while  $i \leq n$  do
6:      $\phi_i \leftarrow \text{CREATEFEATURE}(\zeta_i)$ 
7:      $\tau_b \leftarrow \text{CONCATENATE}(\tau_a, \tau_i)$ 
8:      $\Phi_b \leftarrow \Phi_a \cup \phi_i$ 
9:      $\Phi_b \leftarrow \text{ANALYZEREPETITIONS}(G, \tau_b, \Phi_b)$ 
10:     $\Phi_b \leftarrow \text{IRL}(G, \tau_b, \Phi_b)$ 
11:     $\kappa_b \leftarrow \text{CALCULATELIKELIHOOD}(G, \tau_b, \Phi_b)$ 
12:    if  $\Phi_a$  is  $\{\}$  OR  $\kappa_a - \kappa_b \leq \kappa_T$  OR  $\Phi_a$  is  $\Phi_b$  then
13:       $\Phi_a \leftarrow \Phi_b, \tau_a \leftarrow \tau_b, \kappa_a \leftarrow \kappa_b$ 
14:       $i \leftarrow i + 1$ 
15:    else
16:       $G, h \leftarrow \text{CREATEGOAL}(G, \Phi_a, \tau_a)$ 
17:       $\mathcal{H} \leftarrow \text{APPEND}(\mathcal{H}, h)$ 
18:       $\Phi_a \leftarrow \{\}, \tau_a \leftarrow [], \kappa_a \leftarrow 0$ 
19:    if  $\Phi_a$  is not  $\{\}$  then
20:       $G, h \leftarrow \text{CREATEGOAL}(G, \Phi_a, \tau_a)$ 
21:       $\mathcal{H} \leftarrow \text{APPEND}(\mathcal{H}, h)$ 
return  $\mathcal{H}$ 

```

plays the goals sequentially by its feature vector, and then this sequence is checked to evaluate how much of the criteria are fulfilled. This fulfillment condition is determined by a criterion threshold c_T . A criterion threshold is required since the synthetic agent has no real experience and the human-like agent plays a different level. Furthermore, the criterion of each feature is used to dampen the weights after criterion is fulfilled and to reward goal completion. This supports to distinguish various test goals that have distinctive criteria but similar features. Moreover, the main objective of the agent is to complete its test goal. As goal completion depends on criteria, we reward the agent w.r.t. the completion percentage of the goal. In this study, this additional reward is defined $reward^{completion}$ where $reward$ is a positive real number and $completion \in [0, 1]$. Individual completion is calculated for each criterion and then these values are multiplied to obtain the total completion of the current goal. Therefore, if a single criterion's completion is calculated as zero, then the total is zero as well. Note that after each goal is completed, the current interaction state is reset.

We use the state space, action space, and reward function described in Section IV-A. We have two kinds of agents: MCTS and RL agents. In our MCTS agent, we use knowledge based evaluations in the MCTS to evaluate the states in simulation phase. We use transpositions [44] to share information amongst states, and utilize UCT3 [44] in selection phase of MCTS. In our RL agent, we use Sarsa(λ) algorithm described in Section II-B. Eligibility traces support propagating the calculated long term reward, which is obtained by fulfilling a goal. Lastly, we chose Boltzmann exploration policy [17].

Test Oracle: Generating and executing a test sequence are not enough without determining whether the test fails or not.

Test oracle provides a mechanism that determines whether the software behaves as expected in a test run. Automating the test oracle greatly improves the test execution due to the elimination of manual examination of a test execution result. As our aim is not detecting visual glitches but to find the dissimilarities between the game design and the implementation, we did not use a vision-based oracle. Therefore, we opt to use a model-based oracle that determines fail and success of an execution at each iteration of the game loop [45].

This oracle makes a model comparison using the game scenario graph and the game. This comparison verifies the game transitions using the scenario graph transitions. For example, if the avatar does not possess the key and the door is in play, the game should not be won. However, the model from the game scenario graph cannot be used to catch bugs such as wall collision. Thus, we used additional constraints to catch these bugs such as if the avatar dies, it should have collided with the fire sprite; or the position of avatar should not overlap with any wall. The oracle checks if the game state and interactions violate any of the constraints. The model and the constraints are given by the game developer.

V. EXPERIMENTS

A. Experimentation Setup

In this study, three different games are prepared using GVG-AI framework (see Appendix B and C). These games have varying difficulties and their dimensions are 6x7, 8x9, and 10x11. We refer to these games as Game A (6x7), Game B (8x9), and Game C (10x11), respectively. Each of these games has four different levels. These levels differ in terms of layout. Also, the avatar can use the *Use* action only in the first two levels. 6x7 game is quite simple: the avatar has to pick up the key and go to the door. In the 8x9 game, the avatar has to extinguish the fire, pick up the key, and finish the level by going through the door. In the 10x11 game, the avatar has to create the key by combining the key parts, pick up the key, and finish the level by going through the door. In the 8x9 game, each level has a different game scenario graph.

We use fault seeding to verify our approach. Fault seeding is a technique to evaluate the fault detection rate of software tests and test process [15]. The source code of our games are the VGDL descriptions —not the GVG-AI engine; hence we altered VGDL descriptions while inserting faults. These faults affect the implementation of the game to behave different than the ideal design of the game. We use [46] as a reference to diversify the bugs. We alter the VGDL game description by: removing lines from interaction set, changing the order or the name of the sprites in the interaction, and adding fallacious interactions. We inserted a total of 45 bugs into the VGDL scripts of the games. During scoring, if there are multiple occurrences of the same bug, it is counted as one.

We collected a total of 427 trajectories from 15 different human participants that have different gaming and testing experience. During the data collection, for each game, we stated the rules of the game and their goals, but there were no directions. Testers were able to play the same level any number of times and could even go back and forth between

levels and games. There were tutorial levels for players to get used to the controllers and the environment. We should state that except the tutorial levels, all of the games included bugs. We used the GVG-AI framework to collect trajectories, and for each game, a total of 118, 173, and 136 trajectories are collected, respectively. It should be noted that 8x9 has more game scenario paths than the other games; as a result, testers executed more tests on this game.

We applied MGP-IRL to these trajectories and used Maximum Likelihood Inverse Reinforcement Learning (MLIRL) [35] as the IRL algorithm since it internally calculates the likelihood of the trajectory and it is robust to slight mistakes or noise. We chose three different likelihood thresholds κ_T : 0.0, 0.5 and 1.0, note that MLIRL calculates log likelihood. We compared these three different threshold values to assess the effectiveness of the MGP-IRL. Threshold of 0.0 is our proposed approach for finding weights and threshold of 1.0 is close to using MLIRL. We used the following parameters for MLIRL described in [35, Algorithm 1]: $M=20$, $\beta=5$, $T=0.01$. For each game, for each tester, and for each level, a human-like test goal is learned using the collected trajectories from the other three levels. Thus, human-like agents generate more tests than the original human tester.

For the synthetic test goals, we entered the game scenario graph and the sprites. For our 6x7, 8x9 and 10x11 games, synthetic agent produced 28, 234, and 88 different test sequences, respectively. We used all path coverage since these game scenario graphs do not contain any loops. Lastly, to investigate the effect of modifications on bug finding, a baseline agent is created. The baseline agent uses test goals directly generated from the graph.

We did not use off the shelf testing tools record/replay, test automation frameworks, and monkey testing in our comparison. The record/replay tools will fail even when the direction of the avatar is different. Test automation frameworks require an expert to manually design scenarios, which is not only arduous, but also not scalable. Lastly, monkey testing generates random events to stress test the UI, which is not our test objective. Therefore, COTS testing tools were not adequate in our experiments.

We chose the following parameters for Sarsa(λ): $\gamma=0.95$, $\beta=1$, temporal difference $\lambda=0.8$, learning rate $\alpha=0.03$, and for MCTS: $\gamma=0.95$, exploration term $C_p=3$, rollout depth is 8, 300ms for computation budget on i7-4700HQ using single core. We ran the MCTS 5 times. Our criteria threshold c_T was set to 0.01, goal completion reward was set to 10, and the features that have other than non-zero reward are considered as a singular feature with a reward of -1. We have chosen not to progress an agent if the agent does not accomplish its current goal. As there is no clear definition of a terminal state, we experimented with game lengths 50, 100, 150, 200, 250, and 300 and chose the one that achieved the highest criterion completion. For Game C (10x11), we set direction preference to *All* only to decrease the memory requirements. The running time of Sarsa agent was between a few minutes to 6 hours, depending on the complexity of the goal sequence being played. Lastly, some bugs allow the testers to go out of the intended grid area. Since outside of

the grid was not modeled, we assumed that the agent was interacting with the *Floor* sprite. While training with MCTS and RL, this behavior caused problems such as divergence. Therefore, we assigned a negative reward when the agent tried to explore outside after getting out of the map.

B. Results

In this study, we asked the following questions.

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

To assess bug finding performance, we compared eight different tester groups: original human testers, three human-like Sarsa agents, one MCTS agent with likelihood threshold of 0.0, one baseline Sarsa agent, and one MCTS and one Sarsa agent with synthetic goals. There are two different figures for bug finding performance. The first one uses barplot to compare these groups in each game. Individual agents contribute to the score of a group together, and the total bug count is considered as total unique bugs. The bugs found percentage of MCTS agents is mean of 5 runs. The second figure compares the individual testers in human-like agents and the original human testers. We use violin plot to depict this figure.

Human-like agents' similarity is evaluated using the cross-entropy of human behavior and agent behavior. The trajectory obtained from the human tester is replayed to find a list of interactions $\zeta_{0,\dots,n}^h$, then the trajectory executed by a human-like agent that learned from this human tester's trajectory is replayed to find a second list of interactions $\zeta_{0,\dots,n}^a$. Using $\eta_0, \eta_1, Type, AvatarState$, each list is binned, and the frequency of each bin is used to obtain the cross-entropy of $\zeta_{0,\dots,n}^h$ and $\zeta_{0,\dots,n}^a$. We removed the position and direction components of interactions during comparison as these are highly dependent on the level layout. Lastly, although these do not hold comparison values, the number of actions performed by the agents and the number of splits performed by MGP-IRL are examined. Action length figures are shown using violin plots. Cross-entropy and number of splits figures are shown using box plots with IQR=1.5, except cross-entropy uses log scale [47][48].

Fig. 4 shows the bug finding performances of various agents in each game. We see that humans found all of the bugs in the second game and found 90% of the bugs in the first and the third game. In Game A (6x7), the human-like agents with likelihood threshold of 0.0 and the synthetic Sarsa agent surpassed the human performance and found all the bugs, and the synthetic MCTS agent showed a similar performance to of humans. Moreover, there is a clear difference between the baseline agent and the synthetic agent. The bugs found by baseline agent can be interpreted as number of bugs found by playing only the scenarios specified by the designer. The overall performance difference between the MCTS agent and Sarsa varies from 5% to 10%. In Game B (8x9) the difference between different testers is evident. Human-like agents surpassed the synthetic agent and there is a 10% difference

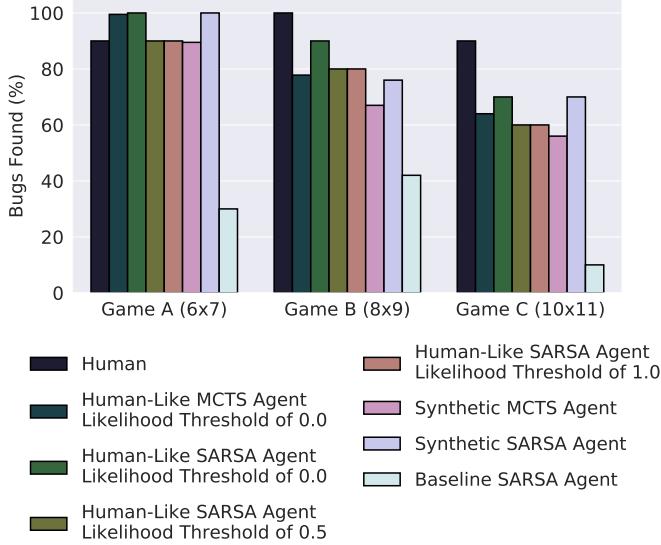


Fig. 4: The Percentage of Bugs Detected by Human Testers and The Generated Agents For Each Game Under Test

between the best human-like agent and the human testers. The performance of the baseline agent is over 40%. In Game C (10x11), we can see the gap between the humans and agents increase, and the synthetic Sarsa agent's performance is on par with the human-like Sarsa agent with likelihood threshold of 0.0. Lastly, test goals used by MCTS agents find fewer bugs in all three games.

1) *Experiment 1: Agents Testing Game A (6x7):* Fig. 5 (Game A (6x7)) depicts the scores of different human-like agents in finding bugs. In a simple game like this, where an agent can generate almost perfect runs, we see that the individual performance of human-like agents is at least the same as or better than individual humans. In Fig. 6 (Game A (6x7)), it is evident that human-like Sarsa agent with likelihood threshold of 0.0 performed the most similar interactions. Moreover, the likelihood threshold affected this similarity in a descending pattern. In Fig. 7 (Game A (6x7)), the human-like MCTS agent with likelihood threshold of 0.0 performed most actions among all agents. Baseline agent was able to finish the game in less than 15 actions. Fig. 8 (Game A (6x7)) displays the number of splits resulted from MGP-IRL: likelihood threshold of 0.0 split more than the others, likelihood threshold of 1.0 agent split the lowest and most of the times, it considered the trajectory as a whole. Lastly, the total bugs found —of all 5 runs— of human-like MCTS and synthetic MCTS are 100%.

2) *Experiment 2: Agents Testing Game B (8x9):* In Fig. 5 (Game B (8x9)) human-like agents have a higher bug finding mean than of individual human testers, but they were not able to perform on a par with the best human tester. Cross-entropy of interactions are similar amongst agents in this game with the human-like agent with likelihood threshold of 0.0 leading, which can be seen in Fig. 6 (Game B (8x9)). Since one of the sprites was missing in the last level of this game, retargeting was not ideal, and the overall cross-entropy is higher compared to Game A (6x7). In Fig. 7 (Game B (8x9)), we see that the

synthetic MCTS agent executed most actions among all agents and had a distinct shape than all other agents. The baseline agent executed at most 40 actions. Fig. 8 shows that (Game B (8x9)) human-like agent with likelihood threshold of 1.0 performed the least amount of splits while human-like agent with likelihood threshold of 0.0 divided the trajectory the most. Lastly, the total bugs found —of all 5 runs— of human-like MCTS is 90% and synthetic MCTS is 76%.

3) *Experiment 3: Agents Testing Game C (10x11):* As seen in Fig. 5 (Game C (10x11)), human-like Sarsa agents improved the overall performance of individual agents, except one tester. Human-like MCTS agents has lower mean values compared to all testers. Fig. 6 (Game C (10x11)) shows that the mean of cross-entropies Sarsa agents are below 0.5 and quite alike when compared to other games. In Fig. 7 (Game C (10x11)) we see that all of the agents executed more actions than previous games. Fig. 8 (Game C (10x11)) reveals that the number of splits for each agent increased when compared to other games. Lastly, the total bugs found —of all 5 runs— of human-like MCTS 90% and synthetic MCTS are 80%.

VI. DISCUSSION

In this paper, we presented a technique for capturing tester behavior, namely interaction state, and introduced two different strategies to generate test goals for agents: synthetic and human-like. We compared the bug finding performance of the agents in three different games and evaluated the similarity of the human-like agents with the original human testers.

Interaction state helped to distinguish previously equivalent states. Interaction state supports this behavior and many more such as attacking all of the walls, covering all empty spaces. Consequently, we were able to model the testing behavior using MDP and MGP-IRL was able to learn tester heuristics from collected trajectories. Note that, these trajectories were collected from games that contained bugs. Though we used the interaction 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 interaction state can engage with these objectives.

Creating a synthetic test goal out of game scenario graph and inserting modifications were valuable since the synthetic agent beat baseline in every experiment (Fig. 4). Baseline agent surpassed half of the individual human testers in Game B (8x9) (Fig. 4 and 5), which has the most complex game scenario graph. This fact reveals that these testers were not able to cover all intended paths of the game, as it is difficult to understand the underlying graph information. Graph coverage provided the path to play the game and modifications guided the agent to numerous paths. Game graph has several advantages as it makes the agent to play each intended scenario and guarantees that these paths are covered, unlike in [8]. Moreover, the generated modifications encouraged the agent to stress the limits of the game. In the experiments, synthetic agent beat every individual tester in human-like agents and most of the human testers (Fig. 4 and 5), which demonstrates the potency of the synthetic agent. Lastly, it provides a flexible mechanism with modifiable coverage criterion and number of

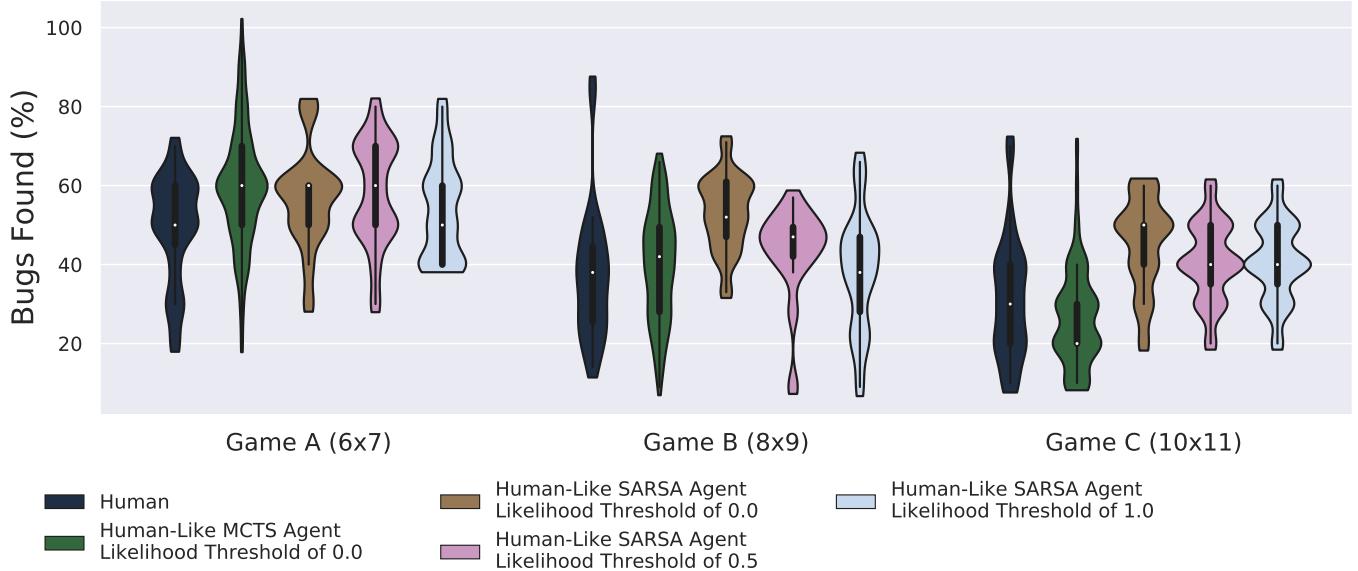


Fig. 5: Percentage of Bugs Detected by Human Testers and The Human-Like Agents For Each Game Under Test

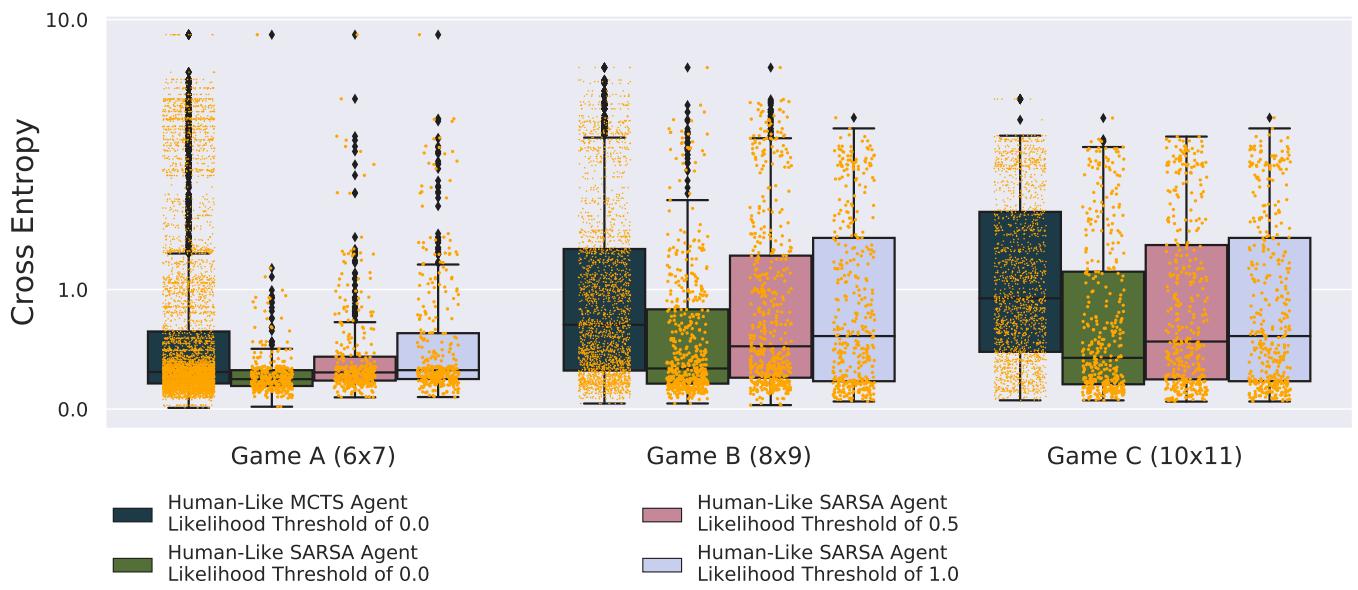


Fig. 6: Cross Entropy of Human-Like Agents For Each Game Under Test

modifications to conduct tests without collecting huge amounts of data. However, synthetic test goals are better utilized by the Sarsa agent compared to MCTS agent, as it found more bugs in all three games (Fig. 4).

Human testers individually were not able to find all of the bugs nor surpass the synthetic agent, but when their performance outcomes were combined, they were able to find most of the bugs and exceed the synthetic agent in two out of three experiments (Fig. 4). This situation is also seen in human-like testers in which the best human-like agent performed the same or better than the synthetic agent in every experiment (Fig. 4). We observe that different human testers traversed different paths of the game and revealed different

bugs. Therefore, it is crucial to find distinct testers. Moreover, as human-like test goals were extracted from these testers' sequences, they benefited from this variance as well. When we compare the bug finding performance of different human-like agents, we observe that human-like Sarsa agent with likelihood threshold of 0.0 is leading both in individual performance (Fig. 5) and in overall performance (Fig. 4). We can attribute this success to multiple goals approach. There are three reasons for this: first, a simple goal is easier to play than a more complex one; second, verifying one goal at a time is better when an agent plays another level since level composition may cause the agent to prematurely skip a feature. Third, the order of test steps in a test sequence is important as this order is designed

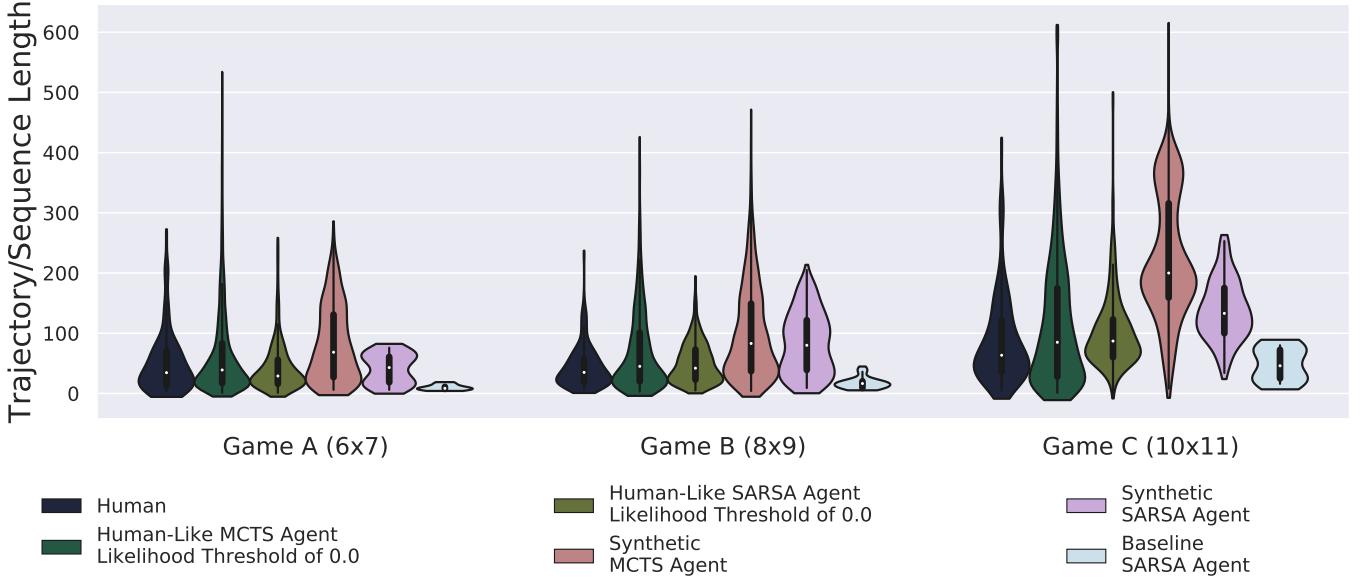


Fig. 7: Trajectory/Sequence Length of Human Testers and The Generated Agents For Each Game Under Test

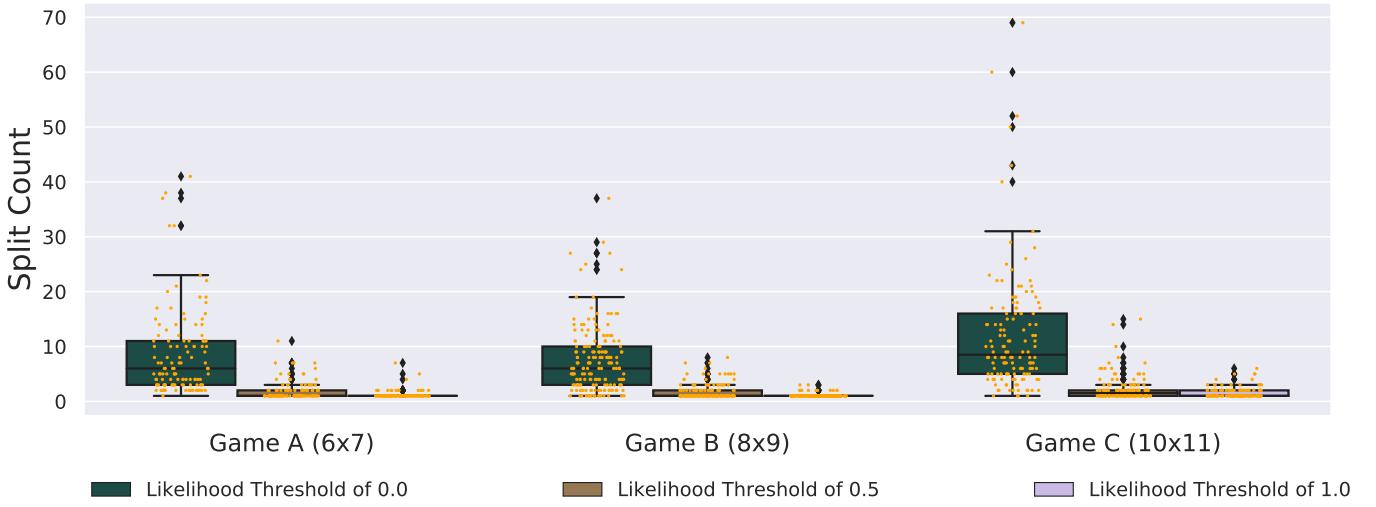


Fig. 8: Number of Splits Done by MGP-IRL of Trajectories Collected From Game A (6x7), Game B(8x9), and Game C(10x11)

for a purpose.

We proposed MGP-IRL to create human-like test goals given tester trajectories. In all of the experiments we see that (Fig. 6) human-like Sarsa agents with likelihood threshold of 0.0 were able to execute more similar interactions to human testers. We noticed that, as the likelihood threshold increases, the mean cross entropy also increases. Bug finding performance of these agents has a non-increasing pattern with the increase of likelihood threshold (Fig. 5). Therefore, we can state that human-like agent with likelihood threshold of 0.0 is both the most human-like and the most successful agent in finding bugs. Fig. 8 shows the number of times the MGP-IRL split the trajectory. In all of the games, likelihood threshold of 1.0 extracted the weights of the features using the whole trajectories.

We used Sarsa and MCTS agents to generate test sequences.

Rewards obtained from certain interactions led the agents to accomplish test goals. The accomplishment of test goals is evaluated using criteria and if the agent successfully fulfilled these criteria, then the agent played the next test goal in the sequence. This approach guided our agent to examine multiple test goals. The mean bug finding performance of Sarsa is greater than that of MCTS (Fig. 4). Synthetic MCTS agent found least amount of bugs amongst all agents. Our manually arranged weights were more fit for Sarsa agent rather than MCTS. On the other hand, one of the human-like MCTS agents with likelihood threshold of 0.0 was able to find all of the bugs, which was not the case for this agent when using Sarsa (Fig. 5 (Game A (6x7))). After careful examination of the bugs, we noticed that the reason behind this difference was due to some fake walls. Since the human agent did not investigate all of the walls, due to stochastic nature of MCTS,

this agent was able to detect this bug in some runs. Moreover, when we accumulated the unique bugs found in all MCTS runs, human-like MCTS agents can find 90% of the bugs in the third game, which is the same as humans'. Therefore, the stochasticity of MCTS is beneficial in testing.

Sarsa and MCTS have inherent bug discovering mechanisms if the agent is guided with proper goals and with right features. In the first experiment (Fig. 4 Game A (6x7)), due to the exploration factor of these algorithms, baseline agent outperformed one of the human testers. If the agent's goal is picking up the key and it is only possible, due to a bug, after attacking the door, then the agent can find this exact sequence. Trajectory plots reveal a difference between testing and game playing—which is performed by the baseline agent. Game A (6x7) has a small board, but there are human testers that executed more than 100 actions (see Fig. 7 Game A (6x7)). In the same game, baseline agent performs at most 15 actions, which is the path length of game scenario graph. In all experiments, MCTS agents executed more actions than Sarsa agents (see Fig. 7). This is expected as Sarsa optimizes the whole sequence while MCTS had to pick an action in 300ms. Hence, the cross entropy results of MCTS are lower in all three experiments (see Fig. 6). The difference between testing and game playing exists in other games as well (see Fig. 7). Another striking factor is that the shape of the synthetic agent is quite different from human testers', but the shape of human-like testers resembles human testers. This reveals that our synthetic approach was non-human, as expected.

Limitations & Challenges: The interaction state increased the number of states explored by our agents, which creates an issue for tabular RL, though interaction state is considerably simple and requires less than 2KB of space for Game C (10x11). This problem can be solved using a function approximator. MCTS can face the same problem only if it utilizes the previously generated game tree.

The main factor affecting human-like performance is the MGP-IRL algorithm. Its greedy approach can be improved with dynamic programming, but this approach will increase the amount of time required to create a test goal. On the other hand, we can direct the human tester to test a game and let the tester segment the trajectory, and then let MGP-IRL find repetition count, direction preference and rewards of the features. However, this approach will apply to an in-house testing rather than open beta-testing. Also, it should be noted that weights found by IRL get better with the amount of trajectory it processes. This approach requires the tester to repeat the same objective in different runs. Moreover, when a tester finds a bug, she will exploit it. When MGP-IRL tries to structure these trajectories, it generalizes these exploited sequences to all situations. This generalization causes a failure in learning the tester behavior. For example, if a tester finds a wall that allows the avatar to pass, the tester tries to carry other objects through there, but human-like agents interpret this wall as any wall. Consequently, the agent will test any wall for this interaction. Moreover, Game C (10x11) consists of a free puzzle unlike Game B (8x9); hence, testers had various solutions to this puzzle. This behavior was neither captured nor repeated easily. This is a limitation of using linear features.

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

VII. CONCLUSION

This paper focused on the problem of creating tester agents. In this regard, we proposed interaction state to capture and execute tester behavior. Furthermore, we presented two approaches to generate test goals: synthetic and human-like. The synthetic test goals are based on sequences from the game scenario graph. These goals are further modified to examine the effects of unintended game transitions. Human-like test goals are learned from human testers' collected trajectories using MGP-IRL. MGP-IRL extracted the tester heuristic in form of features and converted them to goals. We used MCTS and Sarsa agents to play these test goals. These goals directed the agent to different states in the game and agents generated test sequences. These sequences are executed on the game while the oracle checks the game to determine whether the game behaves as expected.

Our results show that the interaction state assisted capturing the human tester heuristic even if the game had bugs and supported MCTS and Sarsa to play the game as testers. The synthetic agent surpassed the baseline agent and most of the individual human-testers and human-like agents. Furthermore, human-like agents, when act together, can compete with the performance of the synthetic agent as well as that of human testers.

We also investigated the bug finding performance of MCTS and Sarsa. We found that the mean performance of Sarsa is better than MCTS, but the stochasticity of MCTS is useful in testing. Furthermore, the combined scores of all 5 MCTS runs showed that human-like MCTS agent competes with humans. Lastly, due to our MGP-IRL algorithm, human-like agent with likelihood threshold of 0.0 behaved similar to the human testers.

To the best of our knowledge, this study is the first to propose human-like tester agents. We showed that these agents can successfully test unexplored levels. Besides, synthetic agent takes model based testing further by introducing the generally acknowledged agent concept in gaming to traditional test techniques. Moreover, once these agents are created, they can test a game any number of times, decreasing the human test effort. Finally, proposing an interaction state enables us to catch the tester strategies and play accordingly.

In the future, we would like to use function approximators in RL agents. Besides, we would like to implement an MGP-IRL that is more robust to random actions. Lastly, we would like to generalize this concept to 3D games and investigate how to model an interaction state in this environment.

ACKNOWLEDGMENT

The authors would like to thank the testers participated in our experimentation.

REFERENCES

- [1] D. Lin, C.-P. Bezemer, Y. Zou *et al.*, “An empirical study of game reviews on the steam platform,” *Empirical Software Engineering*, vol. 24, no. 1, pp. 170–207, 2019.
- [2] B. W. Boehm and P. N. Papaccio, “Understanding and controlling software costs,” *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1462–1477, Oct 1988.
- [3] R. E. S. Santos, C. V. C. Magalhães, L. F. Capretz *et al.*, “Computer games are serious business and so is their quality: Particularities of software testing in game development from the perspective of practitioners,” in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’18. New York, NY, USA: ACM, 2018, pp. 33:1–33:10.
- [4] M. Ostrowski and S. Aroudi, “Automated regression testing within video game development,” *GSTF Journal on Computing (JoC)*, vol. 3, no. 2, pp. 1–5, 2013.
- [5] C. Cho, D. Lee, K. Sohn *et al.*, “Scenario-based approach for blackbox load testing of online game servers,” in *2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, Oct 2010, pp. 259–265.
- [6] S. Iftikhar, M. Z. Iqbal, M. U. Khan *et al.*, “An automated model based testing approach for platform games,” in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2015, pp. 426–435.
- [7] J. Hernández Bécares, L. Costero, and P. Gómez-Martín, “An approach to automated videogame beta testing,” *Entertainment Computing*, vol. 18, 08 2016.
- [8] 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*, 10 2017, pp. 153–164.
- [9] D. Loubos, “Automated testing in virtual worlds,” Game and Media Technology Msc, Utrecht University, 2018.
- [10] S. Gudmundsson, P. Eisen, E. Poromaa *et al.*, “Human-like playtesting with deep learning,” in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 08 2018, pp. 1–8.
- [11] 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*, ser. AIIDE’11. AAAI Press, 2011, pp. 85–90.
- [12] 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*, vol. 7, no. 2, pp. 180–192, June 2015.
- [13] C. Redavid and A. Farid, “An overview of game testing techniques,” 2011.
- [14] E. Adams, *Fundamentals of Game Design*, ser. Voices that matter. New Riders, 2014.
- [15] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.
- [16] N. Tziortziotis, K. Tziortziotis, and K. Blekas, “Play ms. pac-man using an advanced reinforcement learning agent,” in *Artificial Intelligence: Methods and Applications*, A. Likas, K. Blekas, and D. Kalles, Eds. Cham: Springer International Publishing, 2014, pp. 71–83.
- [17] J. G. Kormelink, M. M. Drugan, and M. Wiering, “Exploration methods for connectionist q-learning in bomberman,” in *ICAART*, 2018.
- [18] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, 2015.
- [19] D. Silver, A. Huang, C. J. Maddison *et al.*, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, 2016.
- [20] O. Vinyals, I. Babuschkin, J. Chung *et al.*, “AlphaStar: Mastering the Real-Time Strategy Game StarCraft II,” <https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/>, 2019.
- [21] C. B. Browne, E. Powley, D. Whitehouse *et al.*, “A survey of monte carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, March 2012.
- [22] D. Perez, J. Liu, A. Abdel Samea Khalifa *et al.*, “General video game ai: a multi-track framework for evaluating agents, games and content generation algorithms,” *IEEE Transactions on Games*, pp. 1–1, 2019.
- [23] M. Genesereth and M. Thielscher, “General game playing,” *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 8, 03 2014.
- [24] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [25] T. Schaul, “An extensible description language for video games,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, pp. 325–331, 12 2014.
- [26] T. Machado, D. Gopstein, A. Nealen *et al.*, “Ai-assisted game debugging with cicero,” in *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, 2018.
- [27] 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)*, Aug 2018, pp. 1–8.
- [28] F. Frydenberg, K. R. Andersen, S. Risi *et al.*, “Investigating mcts modifications in general video game playing,” in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug 2015, pp. 107–113.
- [29] D. J. N. J. Soemers, C. F. Sironi, T. Schuster *et al.*, “Enhancements for real-time monte-carlo tree search in general video game playing,” in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, Sep. 2016, pp. 1–8.
- [30] 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*, Aug 2014, pp. 1–8.
- [31] A. Y. Ng and S. J. Russell, “Algorithms for inverse reinforcement learning,” in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML ’00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 663–670.
- [32] P. Abbeel and A. Y. Ng, “Apprenticeship learning via inverse reinforcement learning,” in *Proceedings of the Twenty-first International Conference on Machine Learning*, ser. ICML ’04. New York, NY, USA: ACM, 2004, pp. 1–.
- [33] B. D. Ziebart, A. Maas, J. A. Bagnell *et al.*, “Maximum entropy inverse reinforcement learning,” in *Proc. AAAI*, 2008, pp. 1433–1438.
- [34] M. Wulfmeier, P. Ondruska, and I. Posner, “Deep inverse reinforcement learning,” *CoRR*, vol. abs/1507.04888, 2015.
- [35] M. Babeş-Vroman, V. Marivate, K. Subramanian *et al.*, “Apprenticeship learning about multiple intentions,” in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML’11. USA: Omnipress, 2011, pp. 897–904.
- [36] B. Michini and J. P. How, “Bayesian nonparametric inverse reinforcement learning,” in *Machine Learning and Knowledge Discovery in Databases*, P. A. Flach, T. De Bie, and N. Cristianini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 148–163.
- [37] A. Šošić, M. Zoubir, E. Rueckert *et al.*, “Inverse reinforcement learning via nonparametric spatio-temporal subgoal modeling,” *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 2777–2821, 2018.
- [38] J. Ortega, N. Shaker, J. Togelius *et al.*, “Imitating human playing styles in super mario bros,” *Entertainment Computing*, vol. 4, no. 2, pp. 93–104, 2013.
- [39] 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)*, Aug 2015, pp. 60–67.
- [40] A. Khalifa, A. Isaksen, J. Togelius *et al.*, “Modifying mcts for human-like general video game playing,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI’16. AAAI Press, 2016, pp. 2514–2520.
- [41] S. Devlin, A. Anspokha, N. Sephton *et al.*, “Combining gameplay data with monte carlo tree search to emulate human play,” 2016.
- [42] D. R. Kuhn, R. Bryce, F. Duan *et al.*, “Combinatorial testing: Theory and practice,” in *Advances in Computers*. Elsevier, 2015, vol. 99, pp. 1–66.
- [43] N. Rhinehart and K. Kitani, “First-person activity forecasting from video with online inverse reinforcement learning,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2018.
- [44] 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*, Dec 2008, pp. 389–395.
- [45] S. Varvaressos, K. Lavoie, S. Gaboury *et al.*, “Automated bug finding in video games: A case study for runtime monitoring,” *Comput. Entertain.*, vol. 15, no. 1, pp. 1:1–1:28, Mar. 2017.
- [46] C. Lewis, E. J. Whitehead, and N. Wardrip-Fruin, “What went wrong: a taxonomy of video game bugs,” in *FDG*, 2010.
- [47] T. A. Caswell, M. Droettboom, J. Hunter *et al.*, “matplotlib/matplotlib v3.0.2,” Nov. 2018.
- [48] M. Waskom, O. Botvinnik, D. O’Kane *et al.*, “mwaskom/seaborn: v0.9.0 (july 2018),” Jul. 2018.

APPENDIX A
REALIZED NODES OF GAME SCENARIO GRAPH

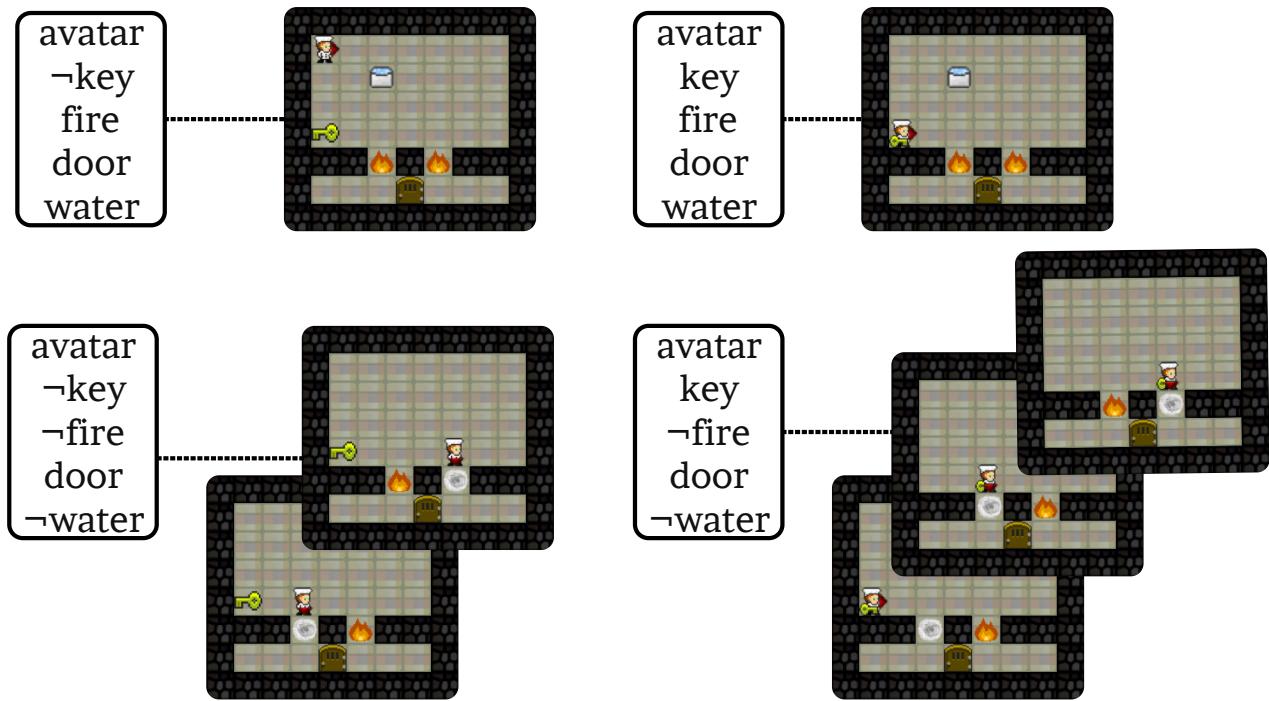
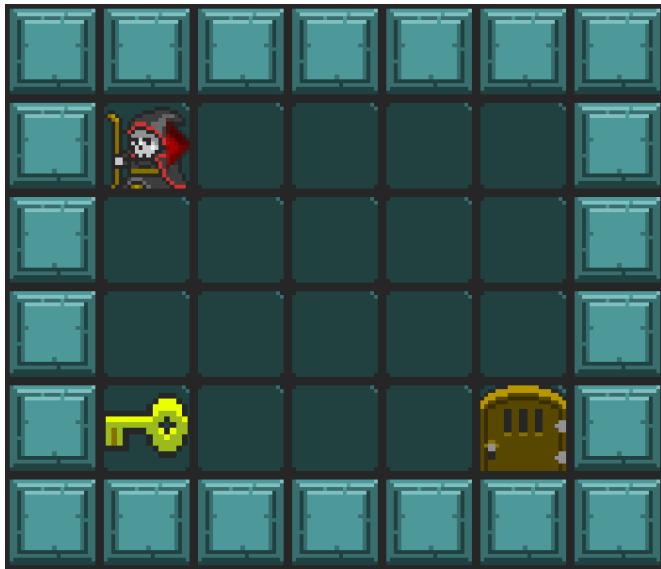


Fig. 9: Realized Nodes of Game Scenario Graph from Fig. 1.

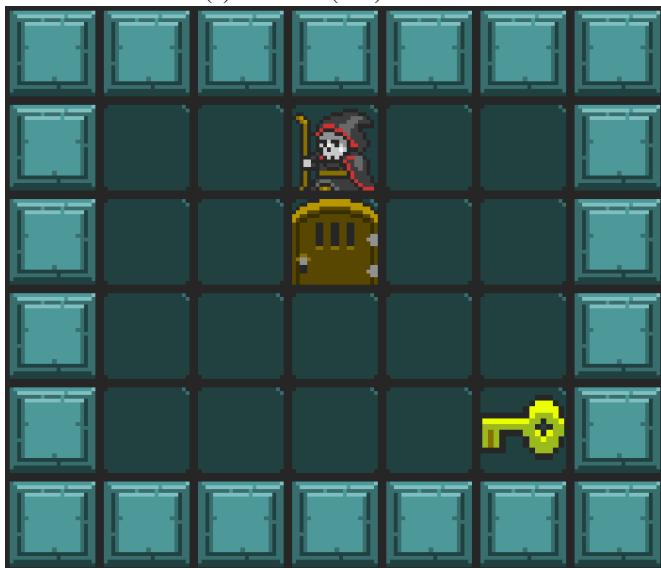
APPENDIX B
GAMES



(a) Game A (6x7) Level 1



(b) Game A (6x7) Level 2

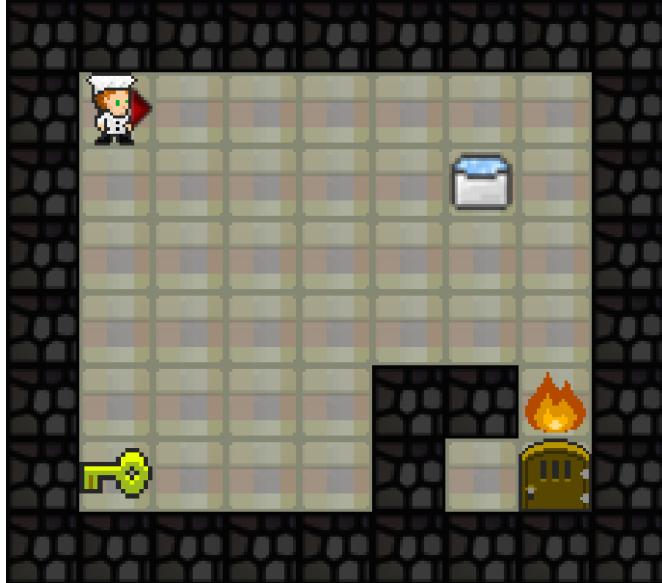


(c) Game A (6x7) Level 3

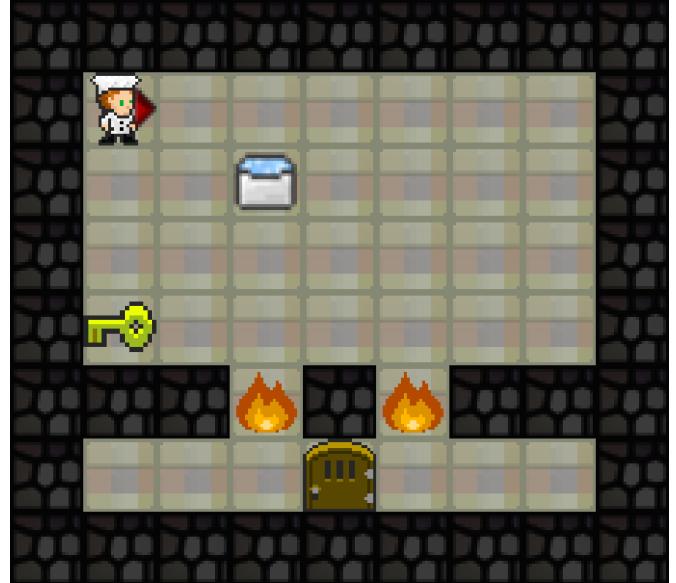


(d) Game A (6x7) Level 4

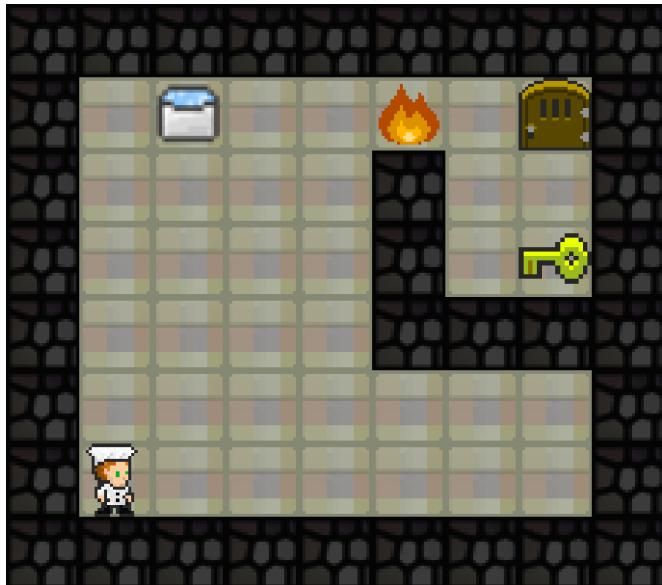
Fig. 10: Game A (6x7) Levels, Representing the Start of the Game



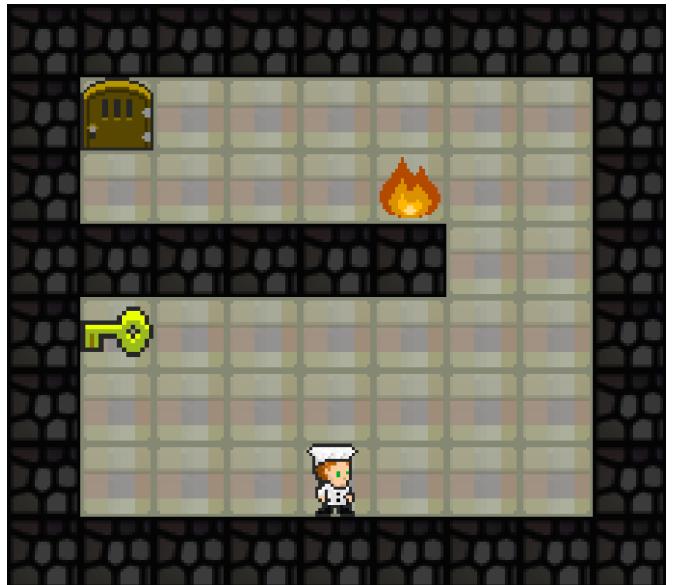
(a) Game B (8x9) Level 1



(b) Game B (8x9) Level 2

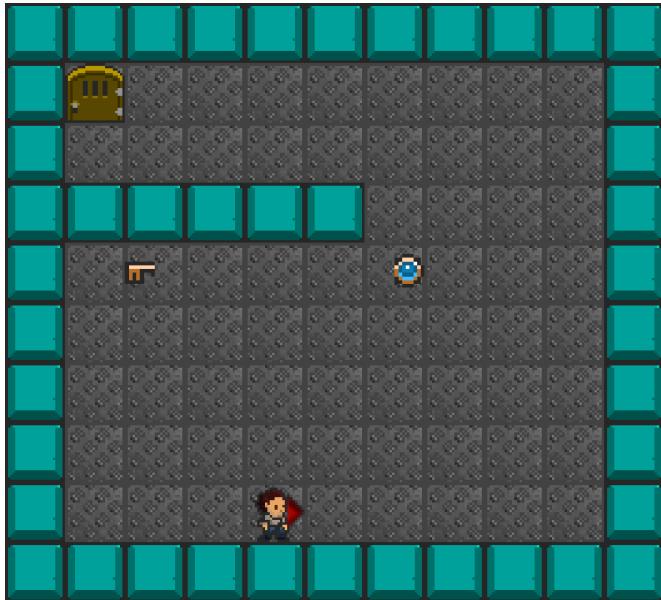


(c) Game B (8x9) Level 3

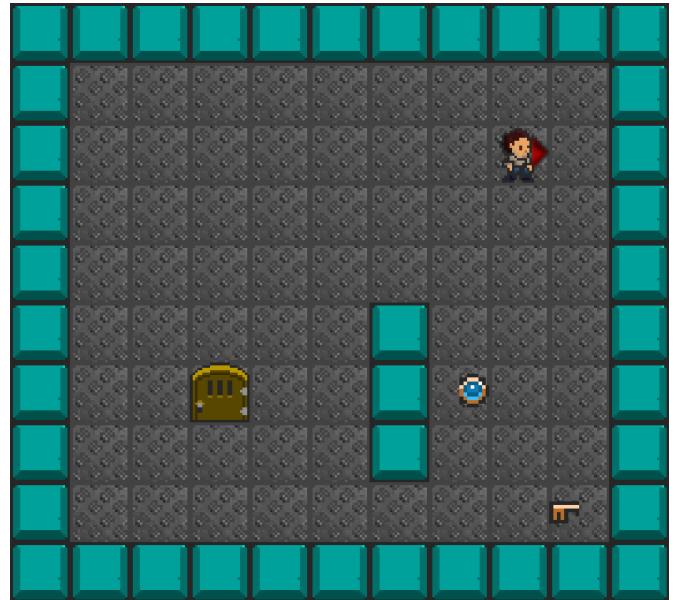


(d) Game B (8x9) Level 4

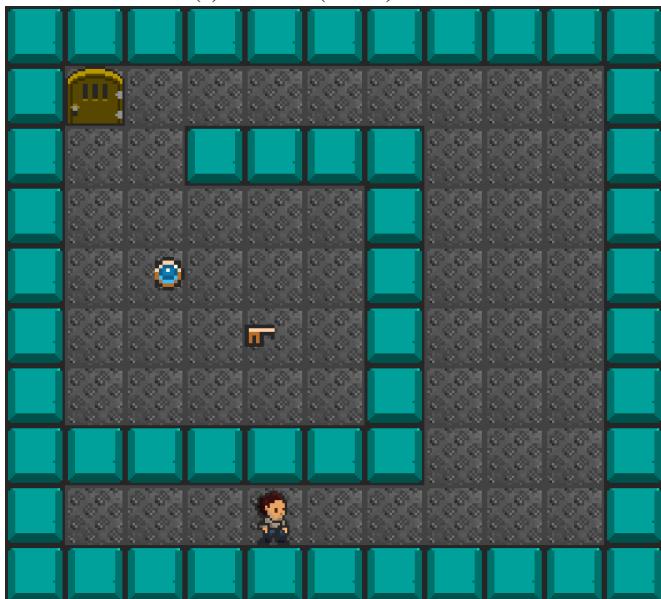
Fig. 11: Game B (8x9) Levels, Representing the Start of the Game



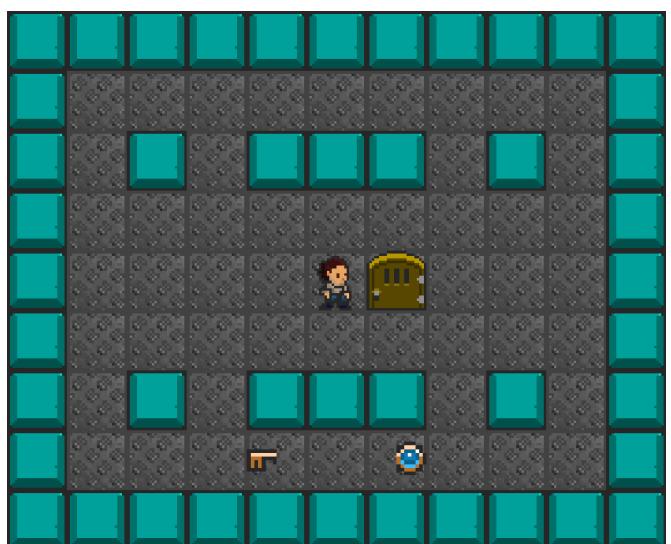
(a) Game C (10x11) Level 1



(b) Game C (10x11) Level 2



(c) Game C (10x11) Level 3



(d) Game C (9x11) Level 4

Fig. 12: Game A (10x11) Levels, Representing the Start of the Game

APPENDIX C

THIRD APPENDIX: VGDL

```

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

```

Fig. 13: VGDL of Game A (6x7) Level 1

```

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

```

Fig. 14: VGDL of Game B (8x9) Level 1

```

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

```

Fig. 15: VGDL of Game C (10x11) Level 1.