Universiteit
Leiden

Master Computer Science

Developing an AI Agent for Automated Playtesting in Roguelike Games

Name: Alex Chang
Student ID: s3768783
Date: 08/08/2025

Specialisation: Data Science: Computer Science

1st supervisor: Mike Preuss
2nd supervisor: Matthias Müller-Brockhausen

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Developing an AI Agent for Automated Playtesting in Roguelike Games

Alex Chang
LIACS, Leiden University
Leiden, The Netherlands
c.chang.6@umail.leidenuniv

***Abstract*—Games with procedurally generated content can pose playtesting challenges for developers due to randomness and resource constraints. This paper presents an AI agent for GONDVAAN, a dungeon crawler roguelike shooter. We discuss the design decisions and technical challenges encountered during development and the effectiveness of different AI architectures for roguelike gameplay. Our implementation evolved from a Finite State Machine to a Behavior Tree that can handle complex decision-making in a dynamic environment. We conducted experiments that evaluate how well the agents complete levels. We found that the agents were successful in completing levels and can have human-like strategies with parameter tuning. Our work demonstrates the feasibility of using AI agents for automated playtesting, reducing development time and providing valuable gameplay analytics for developers.**

***Index Terms*—Roguelike, Playtesting, Finite State Machine, Behavior Tree**

I. Introduction

With the rapid improvement of computer technologies, the video gaming industry is growing extremely fast. Gaming is now not only a thing for kids but a form of entertainment for people of all ages. Besides providing entertainment, video games also have value in scientific research. In fields like Game AI, researchers use games alongside algorithms to conduct experiments. [2]

Traditional video games are often linear, meaning that there is a clear beginning, middle, and end [8], and the player goes through levels carefully crafted by the developers to complete the game. Another genre of video games is roguelike or roguelite games. Inspired by the 1980 game Rogue, a roguelike game features certain characteristics, such as a randomly generated level and permanent death [9]. The player would not be able to the keep progress from previous playthroughs and must start from zero every time. While these restrictions might cause frustrations from the player's perspective, most people enjoy roguelikes for the challenge and mastery of the game's mechanics.

The development of a roguelike game poses an interesting challenge. Developers need a way of making new levels, Procedural Content Generation, while making the game challenging and fun. This can be a tough act to balance without trial and error. There is time and resource costs in finding people to playtest games. With levels being different everytime, gathering data and finding bugs can be hard if the playtesters do not have a lot of time and experience with the game.

Fig. 1. Screengrab of the AI agent playing GONDVAAN

In this work, we will be creating an AI agent to play the game GONDVAAN (Figure 1). GONDVAAN is a 2D dungeon crawler shooter that is procedurally generated.

Our research goals are to:

1) Set up an AI agent to play the game of GONDVAAN
2) Discuss the challenges and decisions when setting up the AI agent
3) Investigate the effects of different strategies for the AI agent

We will now continue with an overview of related works in the domain of game production, AI-based playtesting. Followed by the game description for GONDVAAN, then there will be the setup for the AI Agent with it's results.

We will now have an overview of related works in the field of artificial intelligence (AI) and playtesting (Section II). After that there will be a game description of how GONDVAAN works (Section III), followed by the implementation of the AI agent through various methods (Section IV). There is also an experiment with strategies in AI (Section V) along with results (Section VI). In order to answer the research goals there will be a discussion on the AI design process (Section VII). Finally we will discuss potential future work (Section VIII) and conclude this paper (IX).

II. Related Work

In this section, we will provide an overview of AI and it's relationship with video games, we will also look at how AI is being used for playtesting in game development.

A. Game AI

Artificial intelligence and gaming have a long history together. Games are interesting problems and AI is a method of solving these complex problems. There are many achievements where the AI can play a game better than humans, like Go [16]. With procedurally generated content, can make the problem even harder, the AI has to be smart enough to adapt in new situation. The follow are AI algorithms that were consider for the GONDVAAN's AI agent, but, due to implementation or feasibility, were not used.

Machine learning algorithms like deep learning are popular choices for AI research. Deep learning utilizes neural networks to analyze data and learn complex patterns. In [6], S. F. Gudmundsson et al. is successful in playing the popular match-3 game *Candy Crush* based on human training data. The functions of the AI will also depend on the quality of the training data. The AI would require a large amount of data, which, if produced by humans, would take additional time and resources. As a result, it makes more sense to choose an AI method that can self-play and generate its own data.

Reinforcement learning is a machine learning approach that uses an agent to interact with the game. The advantage of this over machine learning is that it is not dependent on training data. The agent interacts with the environment and develops models; the model is then evaluated and updated based on observations [11]. Reinforcement learning is usually used for 2D games like *Pong*; research has been able to show that it can also be used for 3D games like *Doom*. In [12], G. Lample and D. S. Chaplot developed a deep reinforcement learning model that outperforms the built-in bot and human players. Reinforcement learning methods are more complicated to set up and often have the cold start problem, meaning the agent will have difficulty making decisions in a new environment with no prior data. Additionally, the models from reinforcement learning are not transparent, meaning it is difficult to understand why an agent is making a decision, making it hard to observe strategies.

Monte Carlos Tree Search (MCTS) is an algorithm that finds the best solution by simulating promising actions. MCTS excels at deterministic games such as Go or chess, but it can also be used for games with imperfect information. The algorithm works by having two piece of information, the game rule and the terminal state evaluation, either a win-loss check or score [2]. MCTS is computationally expensive because it runs simulations until a termination state is reached.

Goal-oriented action planning (GOAP) is a system that separates actions and plans. It was originally developed by Jeff Orkinn at Monolith Productions to run the AI in the game *F.E.A.R* [14]. When an AI is presented with a goal, it will try to achieve the problem by finding the best corresponding action. Re-planning occurs when the AI fails to execute an action. For example, if an AI cannot open a door because the player is blocking it, the AI will try other actions such as kicking the door or diving through a window. This system enables dynamic problem-solving for the AI. While the goal is to have the AI perform strategies in GONDVAAN, there may not be that many different ways to solve the game's problems. If a player is stuck in Gondvaan, it usually means they have to either explore more or clear obstacles. In this case, implementing a GOAP system would be overkill.

B. Playtesting

Playtesting is an integral process in the game development cycle. This step allows the developer to gather feedback and see if elements of the game are working as intended. There is now an emerging field that focuses on the topic called Games User Research (GUR); this field intersects the disciplines of Human-Computer Interaction (HCI) and Game Development. Although there is no definitive list of playtesting techniques, GUR has identified different methods such as interviews, A/B testing, and game analytics. GUR aims to research different methodologies in game design that help assess the player experience (PX). The goal is to then share the best practices for practitioners and developers [3].

There is no doubt that having more playtesting can help make a game better; more iterations can lead to more improvements. However, video games are a software product, and a common consideration in development is the amount of time and resources allocated to them. A game made by a large gaming publisher could have a department dedicated to PX. An indie developer, on the other hand, does not have the resources to conduct various trials and interviews. Therefore, developers need to be more mindful about the purpose of the playtest and gather useful results to prevent an inefficient use of time and resources [4].

C. AI methods for playtesting

AI methods can be used to aid developers in the playtesting process. Researchers have tried to use AI-based agents to play the board game *Ticket to Ride* [5]. F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen had different heuristic-based AI agents play against one another on different *Ticket to Ride* maps to see if optimal strategies emerged. This approach combined the researcher's knowledge of the game to have a foundation for the heuristics, and the principles of playtesting to find unexpected strategies. At the end of the work, two failure states that were not covered by the rulings were found, and they found that each map had different optimal agent configurations. In our work, we will attempt to inject strategies to see if we can influence AI behavior.

MCTS has been used for playtesting in a procedurally generated game called *MiniDungeons* 2 [13]; C. Holmgård, M. C. Green, A. Liapis, and J. Togelius used a procedural persona to play a deterministic turn-based roguelike game. They found that different personas were sensitive to different levels of patterns. While this is promising, the movement and combat for GONDVAAN are more complicated than *MiniDungeons*

2, meaning that simulating actions could be more expensive. As a result, this method is not the top choice.

Deep learning also has be shown to have applications in the playtesting field. In the works by [6], S. F. Gudmundsson et al. successfully created a human-like player for the *Candy Crush*. They used deep learning to train on existing player data, and it was then able to correct predict how well a player was going to score. This is important because this method was shown to be less computationally expensive than the Monte Carlo tree search method. One thing to note is that match-3 games are less complex than our procedurally generated 2D shooter, meaning that it would be more difficult to train our model. In a game where the map size and configuration are different every session, it would be difficult for the neural network to extract meaningful strategies from past levels without the risk of overfitting to the specific level.

III. Game Description

GONDVAAN is a roguelike, procedurally generated dungeon crawler shooter game developed by Vincent Prins. Unlike a traditional roguelike, the player is offered a selection of three generated dungeons and weapons at the start of each level. The goal of each level is to survive the dungeon and reach the exit, shown as a ladder in-game. In Figure 1, the ladder can be seen top right of the player. Every dungeon is populated with various enemies that will shoot at the player, there are also objects like obstacles, gems, and health packs [1]. Following the rules of a roguelike, the player can continue to advance through the generated dungeons as long as they do not lose their only life. Another aspect of the game is the dynamic difficulty settings. GONDVAAN has an AI model that adjusts the action intensities of the procedural level as the player advances through levels. These add complexities that will make it challenging to set up an effective AI agent. The following section will go through the different aspects of GONDVAAN and discuss how each component relates to the AI agent.

A. Dungeons & Weapons

Three dungeons are procedurally generated at the start of each level. These dungeons consist of 30 x 30 tiles and have three styles: empty, wall, and floor tiles [1]. A dungeon can be defined as the space between the walls of a level. The dungeon is then populated with various objects and entities for the player to interact with. We will go in-depth into each object below. The player gets a choice between three dungeons, where they can have a small preview of the layout of the dungeon.

In addition to the dungeon selection, the player will also have a choice between three weapon types. There are preview screens for each weapon, showing how the bullets shoot, also showing how the player will recoil when using the weapon. For a human player, the preview screen helps determine which weapon to choose. For example, a player can choose a weapon with a larger bullet spread that makes hitting the enemy easier. For an AI, on the other it can be hard to compare these weapons, although there are parameters of the weapons available, it would be complicated for an AI to determine the effectiveness of a weapon over another, given the different enemy types and random dungeon sizes. For the purposes of this work, the weapon will have constant parameters to ensure that the AI will have the same weapon in each experiment.

B. Characters

In each dungeon, there is the player and the enemies. There are four different non-player character enemy types in GONDVAAN, each with a unique style of gameplay to make it challenging for the player to traverse the level. Each enemy type has different behavior cycles. For example, the dodger will attempt to dodge incoming bullets, and the shy inquisitor will try to escape if damaged [1]. For this paper, we will not attempt for our AI agent to understand the individual behaviors of the enemies and form different strategies. We will use the behavior framework of the enemies as inspiration for how our AI can behave. In the previous work, Prins mentioned that players seemed to spend more time in combat on more difficult levels; we can use this insight to make our AI agent more flexible.

C. Objects

There are different types of objects in Gondvaan, some are crucial to progress the game, like the ladders, and other objects, like chests, can powerup a player to make combat easier.

1) Ladders: When a dungeon is generated, one ladder will be placed in the level. When the player reaches and interacts with the ladder, the level is completed, and the player is taken to a scene to select the next dungeon and weapon. This can be considered the most important goal of the game. The game does not have an ending; the player can continue playing until they lose all their health in a level. For the AI agent, then, being able to detect and effectively reach the ladder will be its priority in every level. We also need to think about how an AI can detect and ”interact” with the ladder. In Section IV-D, we will go over the different methods we used to reach the ladder.

2) Obstacles & Doors: Obstacles come in different sprites and, like the name suggests are designed to block the player's path. Obstacles can be destroy with bullets from either the player or the enemy. Since bullets are a scarce resource in GONDVAAN, the player must manage their ammo whilst shooting. This is crucial because sometimes the ladder is in a space closed off by obstacles. The challenge for the AI agent is similar to the player, however it would be hard for the agent to evaluate whether an obstacle is needed to be destroyed given an unexplored level. In our experiments, our default action is for the AI agent to destroy all obstacles detected, to ensure the agent can explore the entire level.

Doors have two states, unopened and opened. When unopened, the doors have a property similar to the wall, where neither the characters or bullets are able to pass through. The player can interact with the door when near it, similar to the

ladder, and by doing so the door will be opened and behave like the floor. As a player they can choose when they would like to open the door. From a strategic standpoint there is no advantage to leaving a door closed, unless there are enemies beyond the door and the player would like to rest. An open door means the player has access to more areas in a level. For the AI agent the simple logic would be to open all doors it encounters, as opening doors lets the AI agent explore more of the map. However, having more access to the map also means potential for more danger.

3) Items: There are four types of items in GONDVAAN. The first two are consumables that are used when the player walks through them. A health pack will be used if the player walks past it while having less than max health. This is a valuable item because when a player's health reaches zero, the game is over. Collecting a health pack is the only way for a player to regain lost health. Another consumable is the ammo pack; this item adds more ammo to the player's weapon. This is useful because if a player's weapon has no ammo, they can no longer defeat enemies or clear obstacles. For the player, there is no downside to picking up these items, except for the health pack if they are at full health. The AI agent should also share the same behavior, perhaps with an extra incentive to find the health pack if they are below a certain threshold of health. Having this would be crucial to ensure the AI can complete more levels.

The last two items are gems and chests. Gems are dropped when an enemy is defeated, and the players can collect these gems and redeem them for power-ups by interacting with chests throughout the dungeons. The power-ups can make the player move, reload faster, or make them more resistant to damage. Sometimes there are multiple chests in a dungeon; in this case, the player can decide what type of power-up they want based on how many gems they have collected. For an AI agent, this is rather challenging; it is hard for the AI to compare the effectiveness of two or more power-ups. In the current implementation, the AI agent will collect the gems if convenient and redeem power-ups from chests if it passes through and has enough gems.

D. Tutorial Level

Besides the procedurally generated levels, there is also a tutorial level created by the developer. Because each dungeon is generated procedurally, no two game sessions are the same. This is a challenge for the player, but this environment can be especially difficult for an AI agent to get good results. The tutorial level provides a stable testing environment to develop and test the AI agent's functions. It is a good starting point for iterative development.

A good tutorial level should give the player an overview of the mechanics of the game, with some examples of the obstacles a player might encounter when playing. The first level of Super Mario Bros shows different play patterns that a player might encounter [17]. An effective tutorial should help players master the basics of the game and set them up for success.

In GONDVAAN, the tutorial level features a long, horizontally shaped dungeon with doors in between each section. There are five sections in total, and each shows different situations that can happen in a dungeon. Behind the first door is a set of obstacles, in this section the player cannot proceed to the next door unless they use their weapon to shoot and destroy the obstacles. This is a low-stakes environment where the player will not be attacked and can practice with shooting the weapon. They will also see how objects can have health and be destroyed.

Fig. 2. Section 2 of the tutorial level features one enemy reachable by the user and an enemy nest that spawns enemies over time.

The second section features an enemy, an enemy nest, and some walls, shown in Figure 2. In this area, the player is comforted by an enemy that will shoot bullets on sight. The player can use the walls as cover, navigate, and shoot at the enemy to move on. Here, the player will engage in combat, where they have to aim, shoot, and dodge incoming bullets. The player can fail and die at this stage. There is also an enemy nest that is enclosed in a nearby wall. An enemy nest is an object that periodically spawns new enemies; the nest itself cannot be destroyed. If the player lingers long enough, they will see the progress bar above the nest complete and spawn a new enemy. This nest serves only as information for the player, as the newly spawned enemy has no way to interact with the player. For an AI, this section will test its ability to prioritize; the AI must be able to differentiate reachable and unreachable enemies. Combat with cover will also test how the AI interchanges between shooting and moving.

The third and fourth sections showcase the items mentioned in the previous section. The player will first see the health and ammo pack. If the player has sustained damage in the previous section, they will be able to collect and restore some health. By receiving the ammo pack, they can also visually see the ammo count increase on the bottom right side of the screen. The next

section has a chest with a power-up that can be redeemed with gems. While in normal game play, the player starts with zero gems, in the tutorial, the player receives enough gems for the chest even if they chose not to fight the enemy in the previous section. These two sections provide an opportunity for the AI to check its interactions with these items. The AI will have to be able to ignore the health pack when full health, otherwise it will be stuck trying to get an unobtainable item.

The final section has the ladder. By walking up to the ladder, the player will be prompted to press E, the same button used for the chest. By doing so, the player will have completed the tutorial and will be brought back to the main menu. The AI is not able to read and understand text, so it must have other ways to interact with the ladder.

Because the tutorial outlines most of the scenarios that a player will encounter, this will be where the testing for the AI agent will take place. The tutorial will be a crucial tool in the development process for the AI Agent.

IV. Method

The method goes over the implementation of the AI agents. We will go through the different design decisions and iterations. When thinking about creating an AI agent for a game, the first step is to identify the priorities of the AI. For GONDVAAN, there are two main goals: the first is to find and reach the ladder to proceed to the next level, and the second is to survive the level. When a player begins a level, they might not see where the ladder is located, which means they have to spend time exploring the level. While exploring the level, they will encounter hostile enemies and obstacles, and they will then have to combat the enemies, defeating them while retaining their health. With that goal in mind, we have two important functions for the AI agent: ladder detection and combat. The AI agent also has to context switch between the two functions; the AI agent has to be able to execute both functions effectively to complete a level.

A. Finite State Machine

When choosing an AI method, the finite state machine (FSM) was the first method that came to mind. A finite state machine is a mathematical model that describes how different states transition between each other; there is a finite number of states, meaning that for the state machine, there is a defined beginning and end [2]. FSM has been used as a game AI method in video games, mostly as non-player characters. Using FSM for the AI agent means that you can design and control how a character behaves when it interacts with the player or other characters. The advantages of the FSM are that it is simple to design and start. The states and transitions are easy to visualize and debug. The downside, however, is that the system is not adaptable and dynamic. When you have many states that have interconnected conditions, it can be hard to maintain and add new states.

The first iteration of FSM for GONDVAAN's AI agent focused on tackling movement and detection. A script had to be created in Unity that overrides the player controller. The script was implemented as a toggle so it would be easier to debug. The FSM had three main states: detection, movement, and combat. The transitions between these states would be based on what type of object the AI detects, a ladder or an enemy. This state machine was tested in the tutorial level mentioned in Section III-D. Combat was the focus for this AI's iteration; the AI would try to shoot at the enemy if it was nearby. An error occurred when the AI agent tried to shoot the enemy that spawned near the nest, as shown in Figure 2. A human player might understand that an enemy behind the wall is unreachable, but for the AI's detection, the enemy was in close enough range for combat. This was a simple fix, as an additional check was performed to make sure the AI has a line of sight before shooting. However, this meant the FSM had to increase states and, therefore, complexity.

Besides combat, interacting with the ladder also proved to be more complex for the AI. For a human player, a text prompt will appear when a player is near a ladder to press the E key. By pressing E next to the ladder, the level will be completed. To enable the AI to finish a level, a helper function was created to interact with the ladder. There is also a check to make sure the AI is close enough to the ladder. Between the combat issues and the ladder solution, the FSM was becoming hard to manage. Encapsulation was needed to group actions together for easier management.

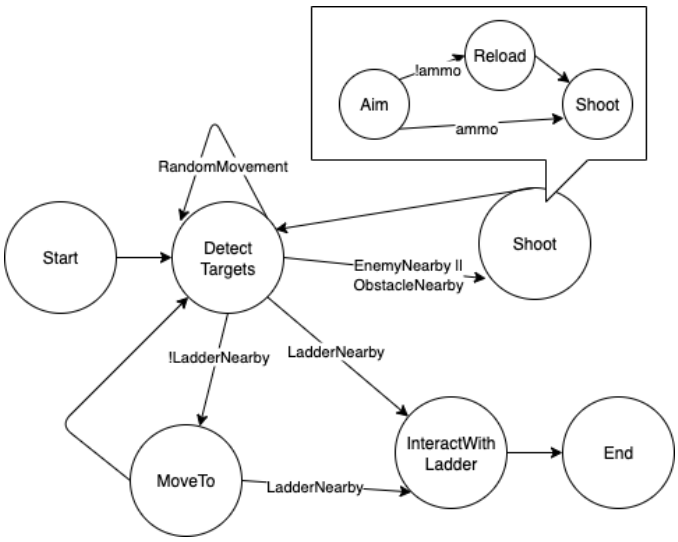

Fig. 3. Visualization of the Hierarchical Finite State Machine. The text bubble represents a super-state, a finite state machine within a state.

A hierarchical finite state machine (HFSM) is a variation of the FSM that includes super-states. A super-state can be thought of as having a finite state machine within a state [7]. Figure 3 shows a visualization of the AI agent in with a HFSM. The main addition is the extension of Shoot's state in the right corner. As mentioned above, there can be conditional checks before the AI shoots at the enemy. The super-states make programming and organizing different functions easier. As the features of the AI agent expand, almost all actions are super-states, which means that there are more states and

transitions to keep track of. Another issue with the AI agent at its current state is that it is too logical. From Figure 3, you can see that the AI moves to another state if the conditions of the transitions are met. If a ladder is nearby, the AI will try to interact with it. This made the AI's action robotic and not human-like; it could not execute complex strategies for playtesting. Although the HFSM is easier to manage than the FSM, more features needed to be implemented, and the FSM was no longer sufficient.

B. Behavior Tree

The behavior tree (BT) is the next AI method used after the finite state machine proved to be difficult to maintain. A BT is a hierarchical model that is connected by nodes. The tree structure works top to bottom with branches and leaves that signify the different actions an AI can take. Unlike states and transitions, the behavior tree has nodes and branches that can be interchanged easily; it also has fallback tactics in case behaviors fail [**?**]. Similar to the states within states in a hierarchical finite state machine, behavior trees can also have sub-trees to group more complex actions together. The interchangeability of the tree's nodes means that it will be easier to add new functions for the AI agent. As we will discuss, the components of the behavior tree also make for intuitive design. It is for that reason that behavior trees are a popular option for modern AI in game development.

The structure of the BT consists of nodes, which are arranged hierarchically from top to bottom, with a root node at the top and child nodes on the bottom. The child node executes an action, in our case, the behavior, then returns its status to the parent node. A node can have three statuses: running, success, and failure [2]. The status is how the nodes communicate with each other; it is also used to determine how the remainder of the tree is traversed. We can order and design the nodes in a way that resembles a strategy for the AI. BTs have four types of nodes that can be used to ease the design process: sequence, selector, decorator, and leaf nodes.

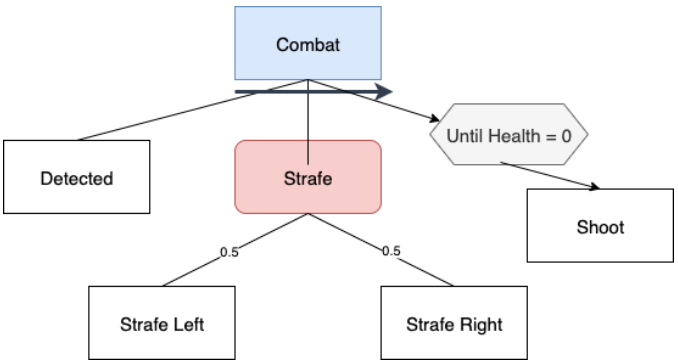

Fig. 4. Example components of a behavior tree. The blue rectangle is the sequence, the red rounded rectangle is the selector, the hexagon represents the decorator, and the remaining rectangles are leaf nodes.

A sequence, like the name suggests, will visit its child nodes in order and can only succeed if all children succeed. If one child fails, then the sequence fails. This is similar to the AND logic in mathematics [10]. This type of node is useful for executing a series of commands. If the sequence for any reason cannot succeed, then the AI will pivot to another behavior. Represented by the blue rectangles in Figure 4, the CombatSequence has three nodes: Detected, Strafe, and Shoot. Once in the combat sequence, the AI will first attempt to detect any enemy. If no enemy is detected, then the AI does not have to continue performing combat-related actions, which allows the AI to quickly switch to another behavior. As stated in the previous section, a simple interaction like finding the ladder can require many conditions for an AI. In a BT, you can create a door sequence that includes detecting, moving towards, and interacting with a door. Most of the actions used by the AI agent for GONDVAAN were sequences. Having a specific sequence for tasks meant we could debug what the AI was trying to achieve very clearly. Sequences are a good foundation for behavior trees because you can chain together multiple logic conditions.

If the sequence is like the AND logic, then the selector is like the OR. The selector node succeeds if one of its child nodes succeeds; if not, then it will try the next one in order. There are two variations of the selector node, the probability selector and the priority selector. In Figure 4, the red rounded rectangle shows a probability selector. The StrafeSelector will have a 50% chance to strafe left or right. (This motion is implemented to try and avoid enemy attacks.) Once one of the options is executed and succeeds, the selector succeeds, the AI can then continue traversing the tree. The probability selector can allow the AI agent to make decisions. Using a probability can also introduce some randomness in the AI's behavior, making it behave more "human-like". In our behavior tree, the probability selector is also used to simulate personas; this can be thought of as AI agents with specific strategies. Using probability, we try to replicate strategies for the AI agent by adjusting the likelihood that the AI will choose certain behaviors over others. Figure 5 shows a simplified implementation of the AI's behavior tree. We will discuss its effects in Section V.

There is also the priority selector; when a child's behavior fails, the selector will try the next child until one returns a success. When visualized, you can see the priority ordered from left to right. In Figure 5, the Utility selector will try the UnstuckSequence first, and if it fails, then the ObstacleSequence will run. The priority selector can be used as a way to ensure the AI has fallback behaviors in case the first node fails [10]. In our use case, the Utility selector is used to clear the way for the AI. When testing the probability behavior tree, we noticed the AI could get stuck easily if all of its behaviors were under the Persona selector. An example would be the AI agent being surrounded by obstacles and unable to move; the AI would have to cycle through behaviors until the obstacle-clearing sequence was chosen. This approach made traversing the map difficult for the AI, so the Utility selector was created to make traversing the environment more consistent. The Utility selector was effective in preventing the AI from getting stuck, letting the AI have space to try

strategies in the PersonaSelector.

The decorator is the third type of node for behavior trees. These types of nodes can add complexity to a child node's behavior; they can be used to invert the results of a node or have the child set to repeat a process a certain number of times. [2] The decorator can be viewed as a while loop or a conditional function in programming terms. The decorator, like all the other types of nodes, can be customized to fit the needs of the system. In Figure 4, the grey hexagon shows a conditional decorator for the Shoot leaf node; once the AI tries to shoot the enemies, it will not stop until the health of the enemies reaches 0.

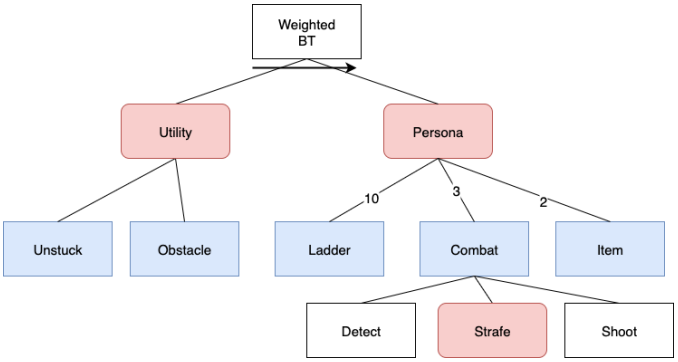

Fig. 5. Weighted behavior tree. The weights in the PersonaSelector lets us change the likelihood of the AI's behavior. This lets the AI have different personas or strategies.

Finally, there are the leaf nodes, which are the actions of the AI agent. A strength of the behavior tree compared to the finite state machine is the reusability of its components. In the following Object Detection section, we will go in-depth into how development time was saved by designing efficient leaf nodes. The previous node types act as the structure of the tree, while the leaf nodes are the specifics of the AI agent.

When switching over the AI agent's implementation from the finite state machine to the behavior tree, the first step was to transfer all the states into nodes. The hierarchical states are similar to sequences, while the smaller states are leaf nodes. The PersonaSelector in Figure 5 was added later in development to facilitate different strategies once all the sequences were transferred and tested. The probability selector was modified to use weights instead, which allowed easier parameterization for later testing.

C. Object Detection

While a human player can look at sprites and discern what they do by interacting with them, an AI agent cannot easily learn the nuances between different items and objects. But with the help of Unity and its tags, we can establish some parameters for the AI agent with object detection.

For Unity, there is the helper function GameObject.FindGameObjectsWithTag, this function returns an array of all the GameObjects with the searched tag [15]. The GameObjects must be declared before compilation, and an empty array will be returned if no GameObjects with the tag is found. While most GameObjects in the game already have tags, some objects, like the door and healthpacks, did not have tags and had to be assigned. For an AI, every object that it needed to interact with needed to have a tag so its function could be categorized in the behavior tree.

A function called CheckObjectDetected was then created to handle all the different GameObjects in the game. There are a total of seven GameObjects: enemy, ladder, gem, ammopack, healthpack, obstacle, and door. This function would be called within the behavior tree sequence of the given GameObject, for example, within the ladder sequence, the AI would first search for the ladder object, then if the ladder was close enough it would try to interact with it, if not, it would try to move the AI in the direction of the ladder. The CheckObjectDetected in this example demonstrates that it is a crucial component for each of the AI's sequences. If the object is not detected, than the sequence fails and the AI can try another sequence. This ensures that the AI is always trying actions that bring it closer to the goal.

D. Pathing

Navigation for the AI agent can be difficult with a lot of trial and error. After identifying the different objects the AI agent can encounter, we can use this information to help the AI traverse the level and find the ladder. A* pathing was chosen for the AI because of its effectiveness in finding the best path.

Before implementing the A* algorithm, we must set up the environment for the algorithm to run on. We first need to create a grid-like structure for the AI., The size of the grid generated is 30x30 tiles, which is similar to the maximum size of a generated dungeon [1]. The tile sizes are also the same size as the in-game character sprite, meaning that it depicts the same range of motion as the AI or player. This is necessary because there are walkways in the dungeon that are exactly 1x1 tiles; if the grid isn't the same, then the AI will have trouble moving through doors. We also limit the grid to a size of 8 tiles surrounding the AI agent to start, then as the AI traverses the level, the grid will be updated with walls as red squares and white icons for objects, like in Figure 6. This dynamic grid approach was used because the levels could change; if an obstacle is destroyed, it could be updated. The small radius in the beginning also helps with computation; this method updates the environment as the AI explores.

A* is a genetic search algorithm that tries to find the optimal path by continuously visiting neighbor nodes until the goal is found. A* calculates a heuristic function to increase computational efficiency [**?**]. In Figure 6, a yellow line is drawn if the AI finds the ladder, and yellow circles leading up to the ladder will be drawn in the debug screen if the AI successfully finds a path with A*. The debug visuals were incredibly helpful in checking if A* was working correctly.

At first, the intuition was that using A*, a well-documented and efficient pathing algorithm, would make the AI agent become too good at completing the first goal. However, as we will see in Section V, the AI will also have to manage its combat effectively to reach the ladder without dying.

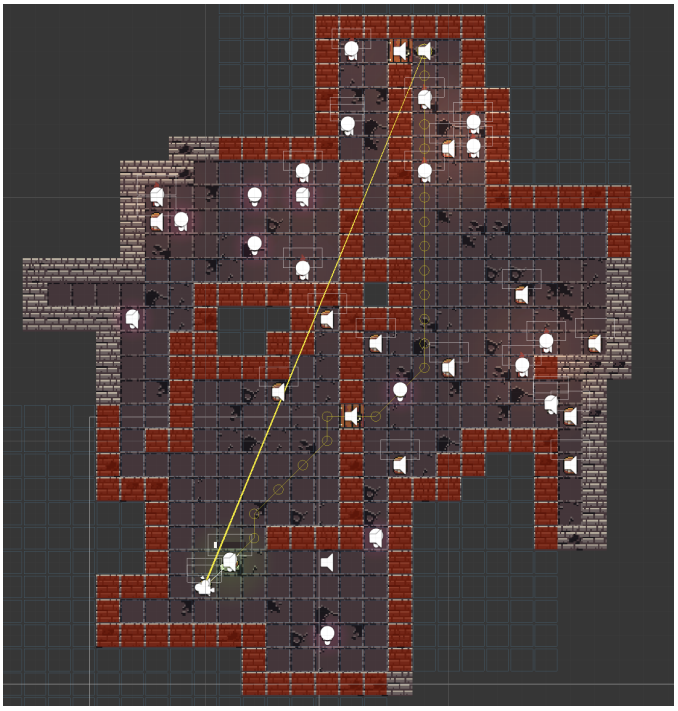

Fig. 6. Debug screen for the AI. The red squares are walls labeled by the A* pathing. The yellow line is the shortest path to the ladder, and the yellow circles are the path to ladder calculated by A*. White icons are other objects in the level.

V. Experiment

In this section, we will discuss our approach to measuring how well the AI agent plays GONDVAAN. For the experiment, we are using the AI agent with the weighted behavior tree method. This version has adjustable weights as parameters to allow for different player personas. Other AI methods were not considered for the experiment as they could not reliably complete a level. The experiment is to have the AI agent complete levels with three different personas: Combat, Speedrun, and Collector. We will then compare the results with human player data gathered from Prins' work [1].

The constraints of the AI agent's functions are as follows. At the beginning of each level selection, the AI agent will randomly choose from the available three dungeons, and for the weapon, the AI agent will also have the same weapon. This reduces some randomness for the experiments. Even though the levels are selected randomly, they are still generated based on the same difficulty settings. This means that the AI will play on different variations of a level that has the same difficulty. Since GONDVAAN is a game that uses procedural generation, we want to make sure the AI agent can complete a variety of levels. However, for the weapon, we decided to have the same weapon across all iterations. The AI uses a single-shot type weapon, which is designed to reduce the complexity of the AI's combat.

There are some constants for the experiment. To align with the human player data, a total of 300 completed levels will be played for each AI persona; a completed level is when the AI successfully reaches the ladder. There is a time limit of 200 seconds per level, which is to prevent a stale state where the AI cannot reach the ladder. The parameters for the AI are the weights for the sequences in the PersonaSelector, as seen in Figure 5. In implementation, the ItemSequence is broken into three separate sequences: health, ammo, and gem. The three personas used and their weights can be seen in Table I. The distribution of the weights came from experience playing GONDVAAN and test runs. An example is that the weights of the LadderSequence for the speedrun personas were originally set to 15, but it could not complete levels because its enemies would kill the AI before it reached the ladder.

TABLE I
Persona parameters for weighted behavior tree.

Parameter	Action	Speedrun	Collector
Ladder	3	9	5
Combat	10	4	4
Health	2	1	8
Ammo	1	0.5	8
Gem	1	0	8

VI. Results

As stated above, 200 complete levels were used for each persona. The three personas had different levels of success rates, with the Action persona having a 96.7%, Speedrun having 79.9%, and the Collector having only a 48.5% success rate. There is also the average time in level, where the Speedrun persona was the fastest at a 48.7-second average, Action at 76.1-second average, and finally the Collector persona at 88.1-second average. All personas had a shooting accuracy of around 55%. These results show that the CombatSequence is most crucial in determining the success rate of a given persona.

All three AI personas completed levels in the range of [1,4], meaning that they did not reach higher levels in the game. This could be because the AI's combat is not sophisticated enough to handle advanced shooting and dodging. Meaning that while the AI can finish a level, it does not have the strategy to maintain health and advance through harder levels.

TABLE II
Pearson correlations between action intensity and game metrics.

Metric	Human	Action	Speedrun	Collector
Interaction	0.53	0.86	0.81	0.9
Time in level	0.42	0.73	0.68	0.76
Time in combat	0.37	0.35	0.28	0.37
Enemies killed	0.63	0.3	0.35	0.24

From table II, we can see the results from the experiments with different personas. The Pearson correlation measures the relationship strength between two variables; it can range from [-1,1], with 1 being positive correlation and -1 being negative. Action intensity is a metric that is used by Prins to quantify the actions when playing. The metrics: Interaction, Time in level,

Time in combat, and Enemies killed, help us understand how the AI is performing in a level.

From table II, we can see that the Personas have a greater interaction than the human player; this could be because the AI agent is making decisions at a much higher rate than the human. We will see, however is that a high interaction correlation does not mean the AI is better at completing the levels. The time in level is higher than the human player across the board. Interesting to note that for the speedrun persona, even though the persona had a greater probability to use the ladder sequence, it did not have a lower time in level correlation compared to the human. This could be because once the AI agent encounters an enemy, it still tries to reach the ladder instead of combat, resulting in death.

From the experiments, it appears that the persona focused on combat was the most successful.

VII. Discussion

For the experiment, we can see that while the AI agent could complete levels in GONDVAAN, it was still not performing enough like a human player for it to be useful for playtesting. However, I still believe there are benefits to this approach. The advantage for having an AI agent for the game is that once the agent is set up you start iterating. For example, by changing the weights, you can have the AI agent perform different strategies in each test. Since both the game and the AI controller exist as scripts, you can also take advantage of the Unity built-in timescale and increase the experiment times, you can have the game run in faster time and complete experiments faster than human players [15]. An argument can also be made that the current actions are not sophisticated enough; the tree can be updated with better sequences.

There are also disadvantages to this approach; the first would be the time required to set up the agent itself. From Section IV, you can see that there were many different iterations of the AI were tried. In Section VI, you can also see that the AI agent requires more tuning. Because of the iterative nature, there will also be room for improvement for the AI.

It was more difficult than anticipated to create an AI agent for the GONDVAAN. Features turned out to be more complicated because there were more things to consider. It's difficult to translate a human strategy into one usable by the AI agent. An example would be implementing the A* pathfinding, I thought that by using an optimized pathing system, it would make the game too easy for the AI, but in reality, it still performed worst than the human player. I suspect this is because the AI also had to handle the combat effectively; otherwise, it could not reach the ladder even if it had the path.

Below are some insights I found while designing the AI agent for GONDVAAN, and could be guidelines for someone attempting to create an AI for another game.

1) What is the winning condition for the game?: The first task is to think about the winning conditions for the game. For GONDVAAN, because it is a Roguelike game, there is no ending, the challenge is to complete each level by reaching the ladder. The scope of winning is then defined by continuously completing each level. There is no time limit in each level; the only way a player could lose is if their health reaches zero. The only way that could occur is if they get hit by an enemy's bullet. That means for an AI, the two main goals are to reach the ladder and survive the enemies, meaning they could either defeat the enemies or avoid the enemies' attacks. With that information, we could work on the most important functions for the AI. From our experiment, we can also conclude that the combat is more important than pathfinding. If AI can handle enemies, then it can have the space to explore. While designing the combat function for the AI came a different problem arose that was solved by the second insight.

2) How do the gameplay mechanics interact with each other?: The second insight for me was to think about how the game mechanics interacted with each other. Unlike humans, the AI needed an accompanying function for each action. When designing the enemy combat, there were two main components: detecting the enemy and deciding if the AI should engage in combat. The first component resulted in the ObjectDetection function, which was adapted to be used for other GameObject detection for the AI. This function was important in helping the AI identify all the elements in the game. The second component became the HasLineOfSight function, which determined if there were obstacles or walls between the AI and the enemy. If there was then the AI would save the ammo and not engage in combat. This was helpful not only to preserve ammo but make sure the AI is focusing on the correct objective, as seen in the tutorial level in Figure 2. By understanding the game mechanics, you can target the most important functions and save a lot of time.

3) Choose the right method: For this work, a lot of time was spent on deciding a suitable AI method. The finite state machine was chosen in the beginning because the states and transitions were easy to define. However, as the actions of the AI increased, it was more and more difficult to manage. The AI ended up being implemented on a modified version of the behavior tree. It could be argued that the behavior tree should have been used in the beginning, but I would say that designing the AI proved to be more complicated than anticipated. Similar to the points made in the previous insight, if I had known more clearly how the game mechanics interacted, I would have known that the structure of the finite state machine was not flexible enough for the AI agent I wanted to create. One of the problems that I encountered was the ever-growing complexity required for the AI to reliably complete each level.

4) Design for iteration: Another important key is to design for iteration. Relating to the right method, you should have a certain degree of flexibility in the AI method. For the behavior tree, it is possible to switch the leaf nodes into new sequences. If this were implemented in the finite state machine it could take more effort when considering how the new state would transition from the previous states.

Designing for iteration also applies to the process of creating the AI agent. After having the foundational actions like movement and combat, the next step is to introduce new features and

test them. By having the tutorial level, I effectively had a test environment for new actions. Because the game is procedurally generated, having a stable tutorial level meant I can test and tune the functions for the AI iteratively. This was helpful as I could build the AI step by step. If I only had new levels to test the AI it would have been difficult to focus on which behaviors to prioritize.

5) Think like a human and like an AI: The final insight would be to practice thinking like a human and an AI. One of the hardest part of this work is coming to terms with how difficult it is to recreate an AI to play a game. As someone with some experience in playing video games, GONDVAAN is not a game that was hard to get started on. For an AI on the other hand is a different story, it was difficult to articulate the objectives and strategies in a way for the AI to understand and execute. When creating an AI for GONDVAAN, the first step was to play the game and analyze the type of strategies that would come to mind while playing. This gave me an insight into how we can approach creating the AI. After playing the games a few time, not only do you get some insights to how the mechanics of the game work, but there are also some strategies that can occur.

The second step is to think like an AI and see if the same strategies can be implemented. By having the weighted behavior tree, there is a certain degree of influence that can be applied to the AI., By having more weights on certain actions like combat, we can try to replicate strategies for the AI while still having some degree of randomness. This flexibility gives us some space for experimentation and testing to improve the AI.

VIII. Future Work

There are two considerations of where this work can go. The original goal was to create an AI agent capable as stand-in for playtesting as a human player. Naturally, the direction is to continue improving the AI agent for GONDVAAN. Another direction is to use the guidelines mentioned in Section VII and try and apply them to another type of game.

A. GONDVAAN

The main discovery from the experiment results is that the AI agent's combat is not as good as expected. The AI needed to have a strategy that was effective enough to finish multiple levels. There is a lot of room for improvement to deepen the CombatSequence. Perhaps more ways of dodging bullets and preserving health, or specific strategies that only occur when the health is low. New personas can also be added to try and solve this problem, and more parameters can be introduced to enrich the AI agent's options.

The functionality of the weighted behavior tree can also be expanded to have dynamic weights. Humans have the ability to switch strategies when playing mid-game. This could also be applied to the AI agent, having a well-rounded persona that changes depending on the situation.

In this work, the AI agent did not have to make decisions for the dungeon, weapon, and chest upgrades. With more data, the AI agent can be trained to make decisions. Whether through human training data or self-play, it could be worth observing if the AI has preferences for certain combinations.

B. Generalization

The guidelines for creating an AI agent can be tested on another game to see if the principles apply. This would be a way of validating this experience report, or perhaps having ways to improve it. The goal would be to find a general AI agent that can be used for playtesting multiple genres of games, which could potentially save developers a lot of time and resources.

IX. Conclusion

In this paper, we presented an AI agent that can play GONDVAAN. We documented the use of two different types of AI methods, from the finite state machine to a modified behavior tree. Through the experiments, we were able to evaluate the effectiveness of the AI agent. Although the combat persona has the highest success rate, it still cannot compare to the human player in terms of effectiveness.

Furthermore, we found that a well-rounded AI agent required more complexity than originally thought. From the results we gathered that the combat was the most important factor to an AI agent's success. This gives us a direction for future improvements.

In conclusion, we found that it was possible to create an AI agent to play a roguelike game. With an effective AI agent, developers can use this as a tool to playtest their games. We hope that further research can be done with this method to help developers and aspiring programmers.

References

[1] V. Prins, "Dungeons & Firearms: AI-Directing Action Intensity of Procedural Levels," Leiden University, 2023.

[2] G. N. Yannakakis and J. Togelius, Artificial Intelligence and Games. Springer Nature, 2025. Available: https://gameaibook.org

[3] P. Mirza-Babaei, V. Zammitto, J. Niesenhaus, M. Sangin, and L. Nacke, "Games User Research: Practice, Methods, and Applications," CHI '13 Extended Abstracts on Human Factors in Computing Systems, pp. 3219–3222, Apr. 2013, doi: https://doi.org/10.1145/2468356.

[4] A. Denisova, S. Bromley, P. Mirza-Babaei, and E. D. Mekler, "Towards Democratisation of Games User Research: Exploring Playtesting Challenges of Indie Video Game Developers," Proceedings of the ACM on Human-Computer Interaction, vol. 8, no. CHI PLAY, pp. 1–25, Oct. 2024, doi: https://doi.org/10.1145/3677108.

[5] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "AI-based Playtesting of Contemporary Board Games," Proceedings of the 12th International Conference on the Foundations of Digital Games, Aug. 2017, doi: https://doi.org/10.1145/3102071.3102105.

[6] S. F. Gudmundsson et al., "Human-Like Playtesting with Deep Learning," 2018 IEEE Conference on Computational Intelligence and Games (CIG), Aug. 2018, doi: https://doi.org/10.1109/cig.2018.8490442.

[7] K.-S. Chang and D. Zhu, "Hierarchical Finite State Machine (HFSM) & Behavior Tree (BT)." Available: https://web.stanford.edu/class/cs123/lectures/CS123_lec08_HFSM_BT.pdf

[8] M. Shepard, "Interactive Storytelling - Narrative Techniques and Methods in Video Games: Linear and Non-Linear," Interactive Storytelling - Narrative Techniques and Methods in Video Games, May 12, 2014. https://scalar.usc.edu/works/interactive-storytelling-narrative-techniques-and-methods-in-video-games/linear-and-non-linear?path=narrative-styles

[9] E. Staff, “The Making Of: Rogue - Page 2 of 2 — Features — Edge Online,” Edge Online, Jul. 03, 2009. https://web.archive.org/web/20121018230001/http://www.edge-online.com/features/making-rogue/2/.
[10] C. Simpson, “Behavior Trees for AI: How They Work,” Game Developer, Jul. 18, 2014. https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work
[11] L. Kaiser et al., “Model-Based Reinforcement Learning for Atari,” arXiv.org, 2019. https://arxiv.org/abs/1903.00374v1
[12] G. Lample and D. S. Chaplot, “Playing FPS Games with Deep Reinforcement Learning,” arXiv.org, Jan. 29, 2018. https://arxiv.org/abs/1609.05521
[13] C. Holmgård, M. C. Green, A. Liapis, and J. Togelius, “Automated Playtesting with Procedural Personas through MCTS with Evolved Heuristics,” arXiv.org, 2018. https://arxiv.org/abs/1802.06881
[14] J. Orkin, “Three states and a plan: the AI of FEAR,” Game Developers Conference, 2006.
[15] U. Technologies, “Unity - Scripting API,” docs.unity3d.com. https://docs.unity3d.com/ScriptReference/index.html
[16] D. Silver et al., “Mastering the game of Go with deep neural networks and tree search,” Nature, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: https://doi.org/10.1038/nature16961.
[17] S. Dahlskog and J. Togelius, “Patterns and Procedural Content Generation,” Proceedings of the First Workshop on Design Patterns in Games, May 2012, doi: https://doi.org/10.1145/2427116.2427117.
[18] X. Cui and H. Shi, “A*-based Pathfinding in Modern Computer Games,” IJCSNS International Journal of Computer Science and Network Security, vol. 11, no. 1, Jan. 2011.