Daedalus & Theseus: An Adversarial Approach to Procedural Generation in Roguelikes

Team Lazarus

Arnav Rustagi *U20220021* | Chaitanya Modi *U20220028* | Mukundan Gurumurthy *U20220056* | Parth Ghule *U20220065*

1 Introduction

This paper introduces a novel adversarial reinforcement learning approach within a roguelike game environment. Roguelikes are video games characterized by procedurally generated levels, perma-death and turn-based gameplay, offering high replayability and diverse challenges. Our aim is to create two interacting RL agents: **Theseus, a player agent**, and **Daedalus, a level generation agent.**

Initially, Theseus is trained to navigate and complete procedurally generated levels. Once proficient, Theseus's performance guides Daedalus. Daedalus learns to generate levels of increasing difficulty, creating an adversarial learning loop. This iterative process enhances both agents' capabilities, with Theseus adapting to the evolving challenges posed by Daedalus's level designs, and Daedalus learning to create increasingly sophisticated and engaging environments.

2 Related work

Work	Focus	Relevance to Our Work
Khalifa et al. (2020)[1]	RL for procedural content generation (Sokoban levels)	Demonstrates the use of RL for level design, inspiring our approach to training Daedalus.
Zhao et al. (2022)[2]	Deep RL for game-playing agents	Provides a general overview of DRL techniques applicable to both our agents
Zheng (2019)[3]	RL in video games	Discusses various RL frameworks and challenges in complex game environments, relevant to our setting.
Goldwaser (2020)[4]	DRL for general game playing	Explores agent adaptation to diverse game environments, a key aspect of our adversarial approach where levels change.

Table 1: Summary of Related Work

3 Problem Formulation

3.1 Labyrinth: The game

Labyrinth is our novel roguelike game, it has the following rules:

1. **Hero:** You command a hero character, characterized by **10 health**, and a gun with a fire rate of **1 bullet per second**, each bullet does **1 damage**
2. **Enemies:** There are 4 different types of enemies the hero can expect to face, each with a different type of behaviour:
    - **Shotgunners:** Close-range specialists dealing bursts of damage, which try to close distance

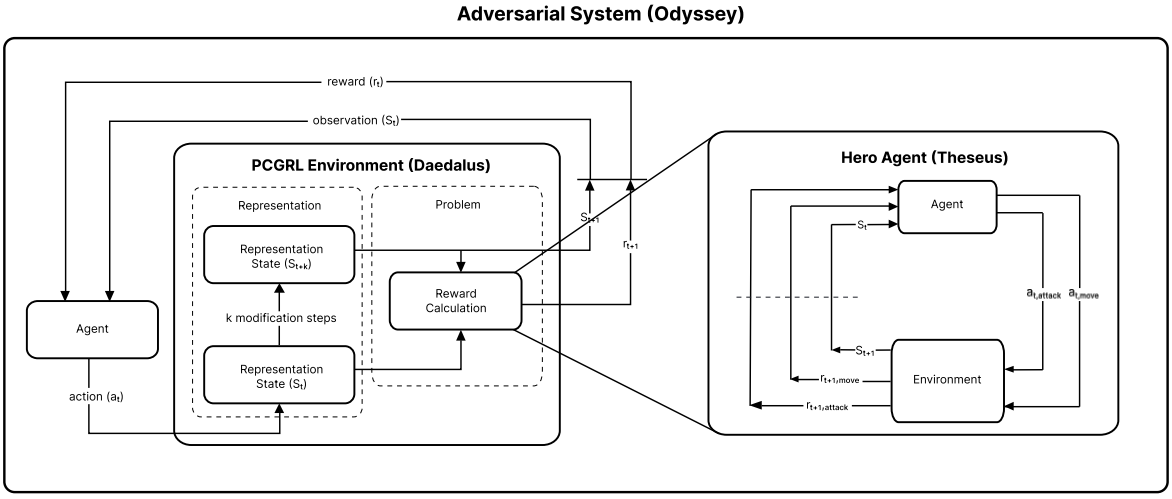


Figure 1: Re-inforcement learning formulation of our entire problem

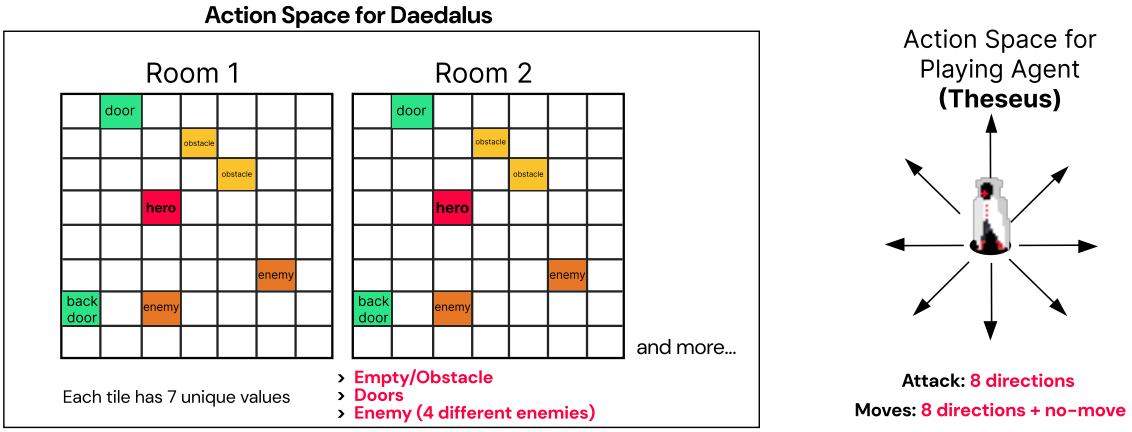


Figure 2: The action space for theseus

- **Snipers:** High-damage, low-mobility long-range units, which try to stay away
- **Quads:** Normal-damage, shoots in four cardinal directions at short intervals, tries to close the distance
- **Gatling:** Low-damage, High-mobility enemies, they shoot a lot of bullets very fast, tries to close the distance

All enemies have **5 health**

Check the code at: https://github.com/team-lazarus/labyrinth

3.2 Theseus: The game playing agent

The game playing agent has the following action space:

1. **8 directions of attacking:** the gun is always shooting and shoots in 8 directions
2. **8 directions of movement + no-move:** the hero can move in 8 directions or choose not to move

The action space together is too dense, so we have made two separate models for attacking & moving, and they are synchronized using symbiotic rewards.

Check the code at: https://github.com/team-lazarus/theseus

3.3 Daedalus: The procedural content generation agent

Our procedural content generation agent is given a pre-generated map made by a standard random walk algorithm, the algorithm initially starts with a map full of walls. Each step has a **0.85 probability** of being an empty cell and a **0.05 probability** of being a wall, and **0.1 probability** of being an enemy. Given the random walked map, the RL agent has to modify tiles, to one of the possible 3 tiles, (0: walls, 1: empty tile, 2: enemies), it modified the tiles using one of the 3 methods:

1. **Narrow:** Simplest way of representing the problem, inspired by cellular automata, the agent is given the current map state, hero state & its location, and the agent is only allowed to modify the tile at the location, and it can choose to either modify or do a `no-action`. So the action space is 4-dimensional (3 modification actions + `no-action`).
2. **Turtle:** Inspired from the turtle graphics program, the agent can move in 4 cardinal directions or modify the tile it is on. The input for this agent is the map state, current location & hero state and output is 7-dimensional (3 modification actions + 4 movement actions(`left`, `up`, `right` & `bottom`)
3. **Wide:** Inspired by Earle (2019) [6] on playing SimCity. At each step the agent has full control on the location and the tile type. The input is only the map state and the hero state, and the action space is `WxBx3`, basically allowing the agent to set any tile on the `WxB` map to one of our `3` tile types.

Check the code at: https://github.com/team-lazarus/daedalus

3.4 Odyssey: The adversarial approach

Odyssey is the adversarial system which uses our 2 models **Theseus** & **Daedalus**, to test if adversarially training these 2 models against each other will provide a better policy.
Here, during each episode, Daedalus generates a map at the start, and Theseus then plays through the episode. Based on metrics such as survival time and level difficulty, feedback is passed back to Daedalus to guide its future map generation strategy.

Check the code at: https://github.com/team-lazarus/odyssey

4 Methodology

4.1 Theseus

Environment

Interaction between the Python RL agent and the Godot game engine occurs via a TCP socket using a custom client/server setup. The process follows a synchronous step cycle: the agent sends a combined movement and shooting action (as a list stringified) to Godot; Godot executes the action, simulates a step, and extracts the resulting game state. This state, including separate `hero_reward` and `gun_reward` signals, a `terminated` flag, and structured information about the hero, enemies, bullets, doors, etc. (positions, health, types), is serialized to JSON and sent back to the Python agent. The agent parses this JSON, uses it to construct graph representations, and prepares for the next action, repeating the cycle. Level resets upon termination (Hero death) are handled within Godot before the next state extraction.

Graph Neural Networks

State representation leverages Graph Neural Networks (GNNs) via `torch_geometric` to process the game's relational structure. The state is modeled as a heterogeneous graph featuring:

- **Distinct node types:** `'hero'`, `'gun'`, `'enemy'`, `'bullet'`, `'door'`, `'wall'`. Each node holds relevant features (position, health, etc.) in its `.x` attribute.
- **Distinct edge types:** `'defeats'`, `'dodges'`, `'shoots'`, `'sees_block'`, `'rev_defeats'`, etc., connecting node types.

Two specific GNN architectures are employed: `HeroGNN` and `GunGNN`.

- **HeroGNN:** Processes the full graph using `HeteroConv` and `SAGEConv` for message passing between different entity types (e.g., enemies informing the hero state via reverse edges). The resulting hero embedding informs movement decisions.

- **GunGNN:** Focuses on the gun-enemy interaction, primarily using `HeteroConv` with `SAGEConv` for messages from enemies to update the gun node's state (`"enemy" -> "rev_shoots" -> "gun"`). This updated gun embedding informs shooting decisions.

This structure enables context-aware actions based on the game state's entity relationships and spatial information.

DQN in Theseus

The Theseus DQN implementation (`AgentTheseusGNN`) uses parallel `HeroGNN` and `GunGNN` agents to learn Q-values for movement and shooting respectively, processing the heterogeneous graph state. Key components include:

- **Target Networks:** Separate, slowly-updated target GNNs (`hero_target_net`, `gun_target_net`) provide stable targets for TD learning, based on the Bellman equation, minimizing oscillations.
- **Experience Replay:** A memory buffer (`self.memory`) stores transitions `(state, actions, next_state, rewards, terminated)` for sampling diverse, decorrelated mini-batches.
- **Independent Updates:** Loss (MSE) is calculated and backpropagated separately for the Hero and Gun components using their specific rewards (`hero_reward`, `gun_reward`). Separate `AdamW` optimizers update the policy network (`hero_policy_net`, `gun_policy_net`) weights.
- **Epsilon-Greedy Exploration:** Random actions are chosen with probability `epsilon` (annealed over time), otherwise greedy actions based on the highest Q-values from each policy network are selected.

PPO in Theseus

The Theseus PPO variant (`AgentTheseusPPO`) employs a dual Actor-Critic structure with separate GNN-based networks for Hero (movement) and Gun (shooting).

- **Components:** Four networks are used: `hero_actor_net`, `hero_critic_net`, `gun_actor_net`, and `gun_critic_net`. Actors learn stochastic policies (action distributions), while critics learn state-value functions.
- **On-Policy Learning:** Trajectories (rollouts) are collected by executing actions sampled from the current actor policies.
- **Advantage Estimation:** Generalized Advantage Estimation (GAE) is used to calculate advantage estimates based on rewards (`hero_reward`, `gun_reward`) and prediction of critic value, balancing bias / variance using `gae_lambda`.
- **Clipped Objective:** Actor networks are optimized using PPO's clipped surrogate objective over multiple epochs (`epochs_per_update`), preventing large policy shifts via the `clip_epsilon` parameter.
- **Critic Training:** Critics are trained concurrently using MSE loss (scaled by `vf_coeff`) against observed returns or GAE targets.
- **Exploration:** An entropy bonus (scaled by `entropy_coeff`) is added to the actor objective to encourage exploration.

Updates are performed using mini-batches from the rollout data before new trajectories are collected.

Contributions

- **Arnav Rustagi:** Development of game & setting up our custom reinforcement learning pipeline, development of the DQN & PPO algorithm utilised by theseus, reward tuning & training.
- **Chaitanya Modi:** Reward tuning.
- **Mukundan Gurumurthy:** Setting up our custom reinforcement learning pipeline, development of DQN algorithm utilised by theseus, reward tuning & training.
- **Parth Ghule:** Architecture & Development of the GNN models for gun & hero.

4.2 Daedalus

Reward Approximator

A neural network reward approximator was developed to accelerate Daedalus's training by replacing a slow, rule-based level critic. This Multi-Layer Perceptron (MLP) predicts rewards for 12×12 maps, enabling efficient and parallelizable reward calculation.

- **Model:** `CriticApproximatorMLP`, an MLP taking flattened $(N, 1, 12, 12)$ maps as input.
- **Architecture:** Two hidden layers `(256, 128 units)` with Layer Normalization, ReLU activation, and Dropout ($p = 0.3$). Outputs a single scalar reward.
- **Training:** Utilized `Mean Squared Error (MSE)` loss with Adam optimizer, using rewards from the rule-based critic as targets. Data was generated online via a Random Walk algorithm; no fixed dataset or augmentation was used.
- **Verification:** Correctness was confirmed by training separate PPO models using both the approximator and the rule-based critic and comparing results.

Map Initialization and PPO Training

Daedalus was trained using Proximal Policy Optimization (PPO) starting from maps generated by a turtle algorithm. This pre-training phase established a baseline performance before transitioning to more complex reward signals.

- **Map Initialization:** A turtle algorithm created base maps by moving randomly and modifying tiles (90% empty, 10% enemy). The number of steps followed a normal distribution ($\mu = 88, \sigma = 16$).
- **PPO Agent (Daedalus):** Modified the initial map over 256 fixed steps using 7 discrete actions (4 movements, 3 tile placements).
- **Reward Signal:** The reward utilised is the difference in Q-value (difference between current and previous map's Q-value) calculated by the reward approximator.
- **Transition:** After initial training with the approximator, Daedalus moved to the Odyssey phase, where rewards were determined by an agent (Theseus) attempting to solve the generated maps.
- **Purpose:** Pre-training provided an initial set of structurally coherent maps for the subsequent adversarial training phase involving Theseus.

Contributions

- **Arnav Rustagi:** Development of Daedalus training system, Implementation of neural-network based reward approximator, Implementation of the PPO algorithm for Daedalus, reward tuning & training.
- **Mukundan Gurumurthy:** Reward function architecture & implementation for Daedalus, reward tuning & training.
- **Parth Ghule:** Implementation of PPO algorithm for Daedalus, reward function architecture & implementation for Daedalus, reward tuning.

4.3 Odyssey

Adversarial Training Framework

Odyssey implements a closed-loop adversarial reinforcement learning framework, integrating our game-playing agent, `Theseus`, and our procedural content generator, `Daedalus`. Odyssey enables dynamic interaction between the two agents and our game engine, `Labyrinth`.

At the start of each episode:

- Daedalus generates a new map using its PPO-based policy.
- This map is injected into the environment, which then initializes the Labyrinth game with the provided layout.
- Theseus—implemented with either DQN or PPO and using Graph Neural Networks (GNNs) for policy and value approximation—attempts to solve the generated map by controlling both movement and gun actions.

Throughout the episode, the environment tracks:

- Rewards,
- Episode length,
- Success criteria (such as wave clearance or termination).

After the episode concludes:

- A detailed summary—including cumulative rewards, episode outcome, and other metrics—is passed back to Daedalus.
- This feedback loop enables Daedalus to adapt its map generation strategy based on Theseus's performance, resulting in the co-evolution of both agents.

Environment Integration

The `Odyssey Environment` extends the base Theseus environment to support dynamic map injection, episode tracking, and feedback passing. It manages TCP communication with the Labyrinth game, serializes and transmits maps from Daedalus, and collects episode statistics for both agents. This design makes sure that the data flow between the agents and the game engine is good, enabling scalable adversarial training.

Contributions

- **Arnav Rustagi:** Architecture of Odyssey's system, integrating Odyssey's code with the pre-existing game code, reward tuning.
- **Chaitanya Modi:** Architecture & Development of Odessey's Python3 adversarial system, reward implementation & reward tuning.
- **Mukundan Gurumurthy:** Architecture of Odyssey's system.

5 Results

5.1 Theseus

Training our theseus' models showed that both DQN and PPO agents effectively learned using the dual GNN architecture, with decomposed control over movement and shooting. Their training demonstrated clear policy improvement, validating the GNN-based framework.

Check the demo for Theseus at: Theseus Demo

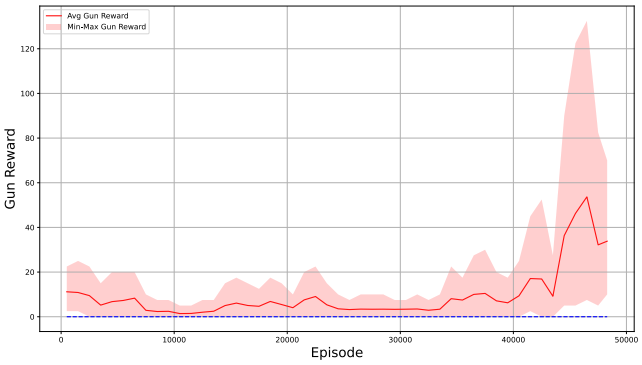
To train both the models together, we had a symbiotic reward structure

- **Rewards for hero:** The hero receives a reward of `+5` for killing an enemy, `+2` for healing, `+5` for clearing a wave of enemies, `+0.5` per enemy for entering their line of sight, and `+10` for clearing all waves.
- **Rewards for gun:** The gun receives a reward of `+2.5` for damaging enemies, `+5` for killing enemies, and `+10` for clearing waves.
- **Symbiotic rewards:** To help the gun-agent co-operate with the hero-agent, we had rewards for the hero-agent for aligning to the enemies, this is called `line-of-sight` rewards. We also have rewards for the hero when the gun-agent `damages` or `kills` an enemy. Both the agents get the most reward for clearing a wave, so they are incentivized to clear the wave, which cannot be done without cooperation.

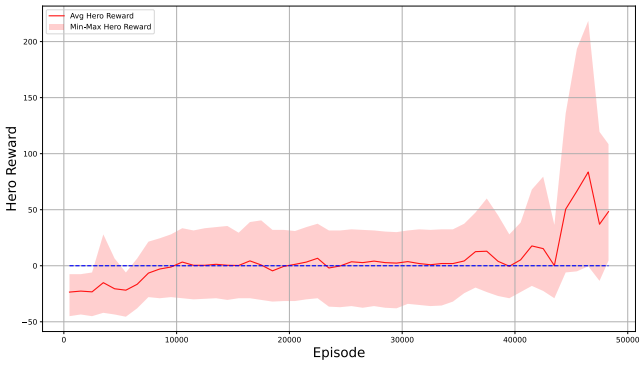
DQN

The DQN agent (`AgentTheseusGNN`) learned an ineffective policy, converging to a suboptimal local minimum. Its strategy failed to promote survival or task completion effectively.

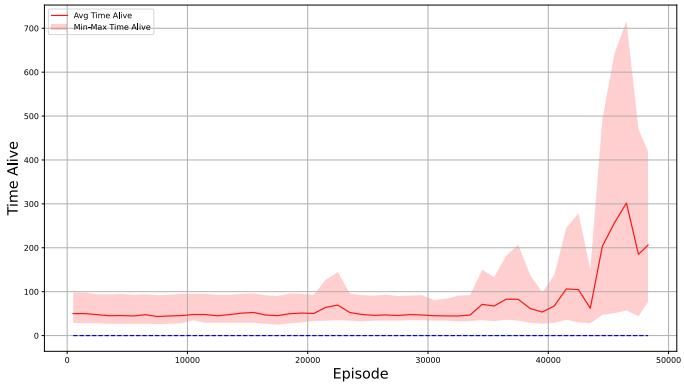
- The agent consistently moved towards a corner.



(a) Gun Reward



(b) Hero Reward



(c) Survival Time

Figure 3: Metrics for Theseus using PPO

- It engaged in continuous firing, often fixed in one direction.
- It lacked sophisticated dodging or adaptive targeting behaviors.

PPO

Conversely, the PPO agent (`AgentTheseusPPO`) successfully learned a dynamic and highly effective strategy resembling the target behavior. This approach proved robust for survival and task execution, confirmed by positive reward trends and increased survival times.

- The agent adopted continuous circular movement around the environment.
- It effectively directed fire inwards towards enemies while maneuvering.
- This strategy facilitated proficient bullet-dodging and enemy elimination.
- Analysis showed the Gun component (shooting) learned faster initially.
- The Hero component (movement) showed more significant improvement in later training stages, mastering coordinated dodging and positioning.

5.2 Daedalus

Check the demo for Daedalus at: Daedalus Demo

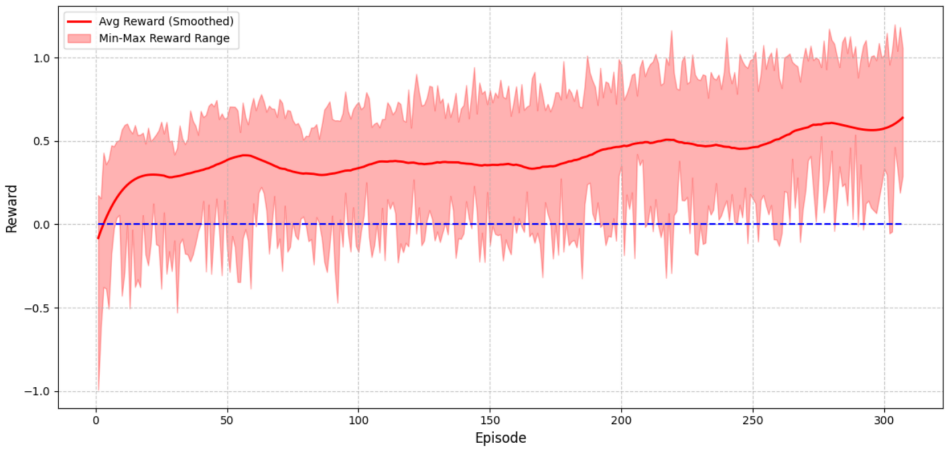
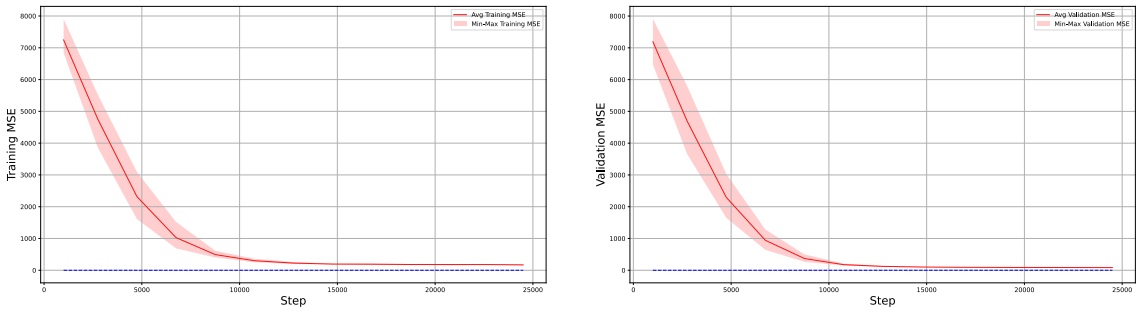


Figure 4: Daedalus average reward over time



(a) Training graph

(b) Validation Graph

Figure 5: Training and Validation MSE of the Neural Critic Approximator

Training methodology

Our initial goal of training Daedalus is to teach it how to modify pre-generated maps (which serve as its input or random seed) so that the maps are both traversable and playable. This allows us to use the pre-trained model for adversarial training within `Odyssey`. We designed our value function with this in mind.

- **Contiguous empty areas:** All empty areas in the map should be traversable by the agent, and no areas should be inaccessible, the agent had a punishment of `-7` for each disconnected tile from the main contigious area. We calculated this area using `breadth-first-search`.
- **Enemies:** We did not want to set any rules for spawning enemies and wanted Daedalus to learn enemy placements while training with `Theseus`. Thus we punish it with a punishment of `-6` for each enemy put over `4` enemies, and `-9` for no enemies placed.
- **Filling the map:** We want Daedalus to fill out the map, so a punishment of `-5` is applied for each traversable tile missing from reaching 50% map coverage.
- **Modifying the map:** We want to encourage Daedalus to modify the map, so we give it a reward of `0.1` for each tile modified.

We primarily trained `Daedalus` with the **TURTLE** representation, each episode started with a random-walked map of size $(12, 12)$, and then Daedalus had to do modifications on top of it, and the actions had to be such that the map is still traversable. The goal of this was to ensure that during Odyssey, `Theseus` can traverse the map and adequately reward Daedalus.

5.3 Odyssey

We will present results for Odyssey in the upcoming presentation.

References

[1] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "PCGRL: Procedural Content Generation via Reinforcement Learning," *arXiv*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2001.09212.

[2] K. Zhao, J. Song, Y. Luo, and Y. Liu, "Game-Playing Agents Based on Deep Reinforcement Learning," *Robotics*, vol. 11, no. 2, p. 35, 2022. Available: https://doi.org/10.3390/robotics11020035.

[3] Y. Zheng, "Reinforcement Learning and Video Games," *arXiv*, 2019. Available: https://arxiv.org/abs/1909.04751.

[4] A. Goldwaser and M. Thielscher, "Deep Reinforcement Learning for General Game Playing," *AAAI Conference on Artificial Intelligence*, vol. 34, no. 2, pp. 1701–1708, 2020. Available: https://doi.org/10.1609/aaai.v34i02.55333.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv*, 2013. Available: https://arxiv.org/abs/1312.5602.

[6] Earle, Sam. (2020). Using Fractal Neural Networks to Play SimCity 1 and Conway's Game of Life at Variable Scales. 10.48550/arXiv.2002.03896.

6 Appendix

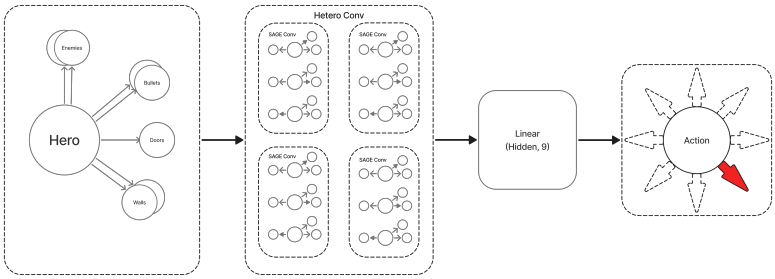
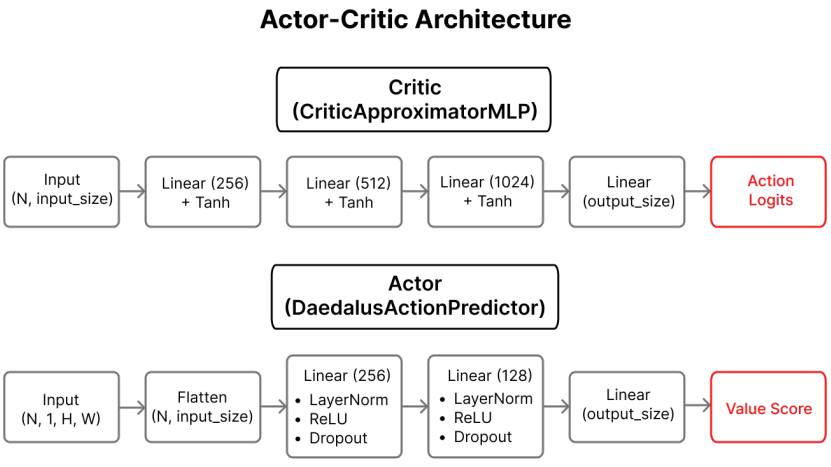


Figure 6: Hero GNN model



Component	Activation	Output
Actor	Tanh	Action Logits (Policy)
Critic	ReLU + Dropout	State Value Estimate

Figure 7: Actor-Critic Architecture for Daedalus