A Modular Framework for Evaluating AI Agents in Procedurally Generated Roguelike Environments

*A report providing the setup and results of a roguelike testing environment.

BS Steyn
(Department of Computer Science)
Stellenbosch University
Stellenbosch, South Africa
21740178@sun.ac.za

***Abstract*—This report provides an exploration of procedurally generated roguelike maps under non-stationary difficulty. Two agents are used to evaluate the PCG framework: (i) PPO with LSTM memory, action masking, and Random Network Distillation (RND), and (ii) a PSO-PPO hybrid. Exploration is operationalised as the mean fraction of unique, reachable tiles visited per map. We evaluate under compound per-level scaling of 2%, 3%, and 5%, with treasure placements providing extrinsic incentives. Metrics include exploration rate, win rate to a fixed depth (discrete), depth-to-failure in endless runs (continuous), actions per level, and sample-efficiency on held-out seeds. Results show PPO(LSTM+masking+RND) achieves higher exploration and degrades more gracefully with scaling than PSO-PPO;**

I. Introduction

This report presents the design and implementation of a modular framework for evaluating AI agents in procedurally generated roguelike environments. By integrating procedural generation techniques for maps and enemy behaviours with a flexible AI architecture, the framework enables the testing and comparison of learning algorithms. Each module is independently swappable, enabling researchers to evaluate the effectiveness of different AI strategies in diverse and unpredictable game conditions. This approach supports experimentation in areas such as reinforcement learning, evolutionary algorithms, and adaptive enemy design within procedurally generated levels, contributing to a comprehensive understanding of generalisable AI in dynamic environments.
While Procedural Content Generation (PCG) was originally introduced to improve user experience through replayability and content diversity, this report focuses on its more recent application: serving as a controlled yet dynamic test bed for evaluating AI agents in adaptive scenarios [7]. The framework in this report is centred on a single procedural content generation (PCG) algorithm designed to create a roguelike environment. This environment can be both discrete and continuous in nature. The opposing agents in the environment are evolved as part of the environmental challenge rather than statically designed entities. Two agents are used to evaluate the framework; the first agent is a PSO-PPO hybrid, while the other is a PPO with Recurrent Policy, Action Masking, and RND Exploration.
This report designs and implements a modular framework for evaluating AI agents in procedurally generated roguelike environments. By leveraging PCG to create diverse and unpredictable maps and enemy behaviours, the framework enables systematic comparison of agent architectures and learning strategies. The focus lies on assessing agent generalisation, adaptability, and robustness in the face of stochastic, evolving environments. To guide the investigation, the report addresses the following research questions:

1) Can the PSG algorithm generate maps with meaningful, diverse connected graphs?
2) In a discrete environment, can the performance of algorithms be compared?
3) Are the results in a continuous environment consistent with those in a discrete environment?
4) Are these obtained results statistically significant?

The remainder of this report is organised as follows. Section II provides a brief background of the topics in this report. Section III provides a review of the available literature. Section IV provides an overview of the methodology used in this project. Section V presents

how the framework is evaluated using the two agents. Section VI provides an overview of the results obtained. Section VII concludes the report. Section VIII provides ideas for future work.

II. Background

This section provides a background for the topics in this report.

A. Procedural Generation

Procedural Content Generation (PCG) refers to the algorithmic creation of game content—such as levels, environments, characters, or rules—with limited or no direct human input. It has become a core technique in game design, particularly in genres like roguelikes, sandbox games, and open-world exploration, where replayability and unpredictability are key.

The origins of procedural generation in games trace back to early titles such as Rogue (1980) [21], which used rule-based algorithms to generate dungeon layouts, enabling unique playthroughs within limited memory constraints. Since then, PCG has advanced considerably, incorporating methods ranging from randomised rule systems (e.g., cellular automata, L-systems, BSP trees) to more advanced techniques like search-based generation, constraint solving, and machine learning models (e.g., GANs, reinforcement learning). Some modern PCG approaches can be classified as Online vs Offline, Constructive vs Generate-and-Test methods and Deterministic vs Stochastic algorithms [19].

B. The Legacy of Rogue (1980)

The 1980 release of Rogue marked a pivotal moment in the history of procedural content generation. Developed by Michael Toy, Glenn Wichman, and Ken Arnold, Rogue introduced algorithmically generated dungeon layouts that offered a new level of replayability, making each play-through unique despite the limited hardware of the time [21]. This design philosophy laid the groundwork for the "roguelike" genre and demonstrated the viability of PCG as a core gameplay mechanic rather than a background optimisation. The game's use of randomness and rule-based generation inspired subsequent research into dynamic level creation and continues to influence modern frameworks that emphasise adaptivity, scalability, and player engagement through unpredictability.

C. Connected Graphs

In graph theory [22], a graph is said to be connected if there exists a path between every pair of nodes. In the case of undirected graphs, this ensures that the entire graph consists of a single connected component, while for directed graphs, strong connectivity further requires that such paths exist in both directions. Connectedness is a foundational property in many algorithmic domains, particularly in path finding, search, and spatial generation tasks. This is relevant for map generation.

D. L-System

Lindenmayer Systems (L-systems) are a class of formal grammars initially developed to model the growth patterns of plants and cellular structures. Introduced by Aristid Lindenmayer in 1968 [9], [10]. L-systems use recursive rewriting rules to generate complex structures from simple initial symbols and rulesets. In the context of procedural content generation (PCG), L-systems have been adapted to algorithmically produce spatial layouts, particularly those requiring self-similarity or branching, such as maze-like environments or organically expanding rooms.

In their basic form, L-systems consist of an axiom (starting string) and a set of production rules that determine how each symbol is replaced in successive iterations. By interpreting these symbols as drawing commands (e.g., move forward, turn left), L-systems can be rendered graphically using turtle graphics or similar traversal schemes. This makes them particularly well-suited to generating maps with recursive, modular room layouts or branching corridors.

The expressiveness and compactness of L-systems allow for highly variable and extensible content generation. When applied to grid-based environments, L-systems can control the placement and connectivity of rooms and corridors, ensuring that generated maps retain consistent structural patterns while exhibiting substantial topological diversity. Their deterministic or stochastic variants can be used to control randomness and adaptiveness in the output, making them suitable for both predefined and dynamic content generation scenarios.

E. Proximal Policy Optimisation (PPO)

Proximal Policy Optimisation (PPO) [16] is an on-policy, actor–critic reinforcement learning method that stabilises policy-gradient updates by constraining the magnitude of each update while retaining practical sample efficiency. PPO optimises a *clipped surrogate objective* using trajectories gathered from the current policy, thereby discouraging overly large policy changes that can degrade performance in non-stationary settings such as procedurally generated levels.

1) Clipped surrogate objective.: Given a policy $\pi_\theta(a \mid s)$ and a reference (behaviour) policy $\pi_{\theta_{\text{old}}}$ used to collect data, PPO maximizes

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\Big(r_t(\theta)\,\hat{A}_t,\ \text{clip}\big(r_t(\theta),\,1-\epsilon,\,1+\epsilon\big)\,\hat{A}_t\Big)\Big],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio and $\hat{A}_t$ is an advantage estimate (often via Generalised Advantage Estimation, GAE). An advantage estimate is an estimate of how much better an action is compared to the average action for a given state. The clipping parameter ϵ (typically $0.1-0.2$) truncates the ratio to limit update size, providing a practical approximation to trust-region methods.

2) Actor–critic losses.: In practice, PPO minimises a composite loss over shuffled minibatches for several epochs:

$$\mathcal{L}(\theta,\phi) = -\mathcal{L}^{\text{CLIP}}(\theta) + c_v\,\mathbb{E}_t\left[\tfrac{1}{2}\left(V_\phi(s_t) - \hat{V}_t\right)^2\right] - c_e\,\mathbb{E}_t[\mathcal{H}(\pi_\theta(\cdot \mid s_t))],$$

where V_ϕ is the critic, $\hat{V}_t$ is a bootstrap target, and $\mathcal{H}$ is an entropy bonus that encourages exploration; c_v, c_e weight the value and entropy terms.

3) Implementation choices.: PPO is typically deployed with:

- **Advantage estimation:** GAE(λ) to balance bias–variance in $\hat{A}_t$.
- **Batching:** Roll out T steps across N parallel environments, then perform K SGD epochs over minibatches.
- **Normalisation/clipping:** Normalise advantages per batch; optionally clip value targets to stabilize the critic.
- **Learning-rate schedule:** Linear or cosine decay; early stopping based on an approximate KL threshold to respect an implicit trust region.
- **Action spaces:** Categorical policies for discrete actions; diagonal-Gaussian policies (with learned log-std) for continuous control.

III. Literature Survey

This section reviews previous work related to procedural content generation, adaptive game-playing agents, and modular AI architectures, with a focus on dynamic game environment applications.

A. Procedural Content Generation

- **Togelius et al. (2011) – "Procedural Content Generation: Goals, Methods and Applications**: This seminal survey paper defines PCG as the algorithmic creation of game content with limited or indirect human input. It categorises PCG along several dimensions: online versus. offline, constructive vs generate-and-test, and deterministic vs stochastic. The authors discuss the use of PCG for various content types (e.g., levels, items, quests) and argue for PCG as a tool not only for scalability but also for enabling personalised and adaptive game experiences. This paper is relevant as it supports the idea of using PCG to dynamically vary game environments and enemy compositions.
- **Summerville et al. (2018) – "Procedural Content Generation via Machine Learning (PCGML)**: This paper introduces the concept of PCG via Machine Learning (PCGML)—using machine learning techniques to learn how to generate game content from data. It surveys ML models such as neural networks, Markov chains, and GANs, and discusses challenges like dataset availability and content quality control. The paper positions PCGML as a way to shift from manually designed rules to data-driven generation. This paper is relevant as it offers a pathway for automated and adaptive generation of maps and enemies.
- **Johnson et al. (2010) – "Cellular Automata for Real-Time Generation of Infinite Cave Levels**: This paper explores the use of cellular automata (CA) for real-time cave-like level generation. Unlike static dungeon generators, the approach continually generates new sections as the player explores, enabling infinite and seamless environments. The method ensures that levels remain playable and interconnected while maintaining the organic look typical of roguelike games. This paper is relevant as it offers a well-suited algorithm for dynamic map generation, ensuring that AI agents face constantly novel spatial challenges during evaluation.
- **Matsumoto & Togelius (2020) – "Rogue-Gym: A New Challenge for Generalisation in Reinforcement Learning"**: This paper introduces Rogue-Gym, a lightweight, ASCII-based roguelike environment designed to test and benchmark generalisation in reinforcement learning (RL). Unlike static environments such as Atari 2600, Rogue-Gym generates diverse dungeon layouts procedurally across

multiple game seeds. The core contribution is its ability to evaluate agent performance on unseen levels, revealing how well RL algorithms generalise rather than memorise training environments. This paper is relevant as it provides an overview of a test suite for evaluating the adaptability of modular AI agents in procedurally generated environments

B. Adaptive Game-Playing Agents

- **Yannakakis & Hallam (2009) – "Real-Time Game Adaptation for Optimising Player Satisfaction"**: This paper presents an early and influential approach to real-time adaptation in games using player-centred AI, aimed at enhancing user satisfaction through adaptive gameplay mechanics. The authors propose a closed-loop system in which the game continually monitors the player's behaviour and adapts various parameters—such as difficulty, pacing, or enemy characteristics—based on learned models of player preference.
  The adaptation system employs neuroevolution, specifically NEAT (NeuroEvolution of Augmenting Topologies), to evolve the parameters of a game-playing agent in response to player feedback. Notably, the system incorporates player input in the form of preferences or annotated physiological data (e.g., arousal, engagement), training a model that links game state features to subjective enjoyment.
  The Experimental domain is a simple shooter game where enemies adapt to maximise predicted player satisfaction. The study finds that real-time adaptation leads to more engaging experiences and demonstrates that machine learning models can predict optimal game configurations tailored to individual players.
- **Justesen et al. (2019) – "Illuminating Generalisation in Deep Reinforcement Learning through Procedural Level Generation"**: This paper explores the critical issue of generalisation in deep reinforcement learning (DRL) by introducing a methodology that leverages procedural content generation (PCG) to expose agents to a broad distribution of levels during training. The authors argue that standard DRL benchmarks often result in agents that overfit to narrow sets of deterministic environments, failing to transfer learned policies to novel scenarios—a major obstacle in creating robust game-playing agents. To address this, the author proposed two training strategies:
  - **Progressive PCG (PPCG)**: Which gradually increases level complexity as the agent's performance improves, forming a form of curriculum learning.
  - **PCG via RL (PCGRL)**: In which a secondary agent learns to generate levels that are challenging and informative for the main agent, creating a co-adaptive training loop.

  Experiments conducted in a simple 2D platformer show that agents trained with procedural variation perform better on unseen levels than those trained on fixed environments. However, the results also highlight that even with PCG, generalisation is far from solved, indicating the need for more sophisticated learning and evaluation strategies

IV. Methodology

This section provides an overview of the methodology used in the various parts of this project.

A. Software Used

This section provides an overview of the various software used in the creation of this report.

*1) **Language and Runtime.**:* All Experiments were implemented in Python. The roguelike environment was implemented in Java.

*2) **Core Libraries.**:*

- **PyTorch** [12]: primary deep learning framework for model specification, automatic differentiation, and GPU-accelerated training. Used to define networks, loss functions, and optimisers, and to dispatch computations to CPU/GPU as appropriate.
- **Matplotlib** [4]: analysis and visualisation. Used to produce publication-quality figures (learning curves, ablations, confidence intervals) directly from Experiment logs.
- **Pygame** [14]: lightweight 2D simulation and interactive rendering. Used to instantiate the game/agent environment, manage the event loop, and provide deterministic frame stepping for evaluation.
- **JPype** [11]: Python–Java interoperability. Used to call pre-existing Java components (e.g., heuristics, parsers, or evaluation tooling) from Python without re-implementation, enabling direct integration with the PyTorch training loop.

*3) **Integration Pattern.**:* The framework exposes a step-based environment API that emits observations and rewards each frame. PyTorch consumes these observations to compute actions and perform gradient updates. Metrics and diagnostics are buffered in Python and

plotted with Matplotlib. Where Java integration is required, JPype launches a JVM within the Python process and binds Java classes as Python objects; these are invoked synchronously during environment steps or post-processing, with data passed via JPype's automatic type conversion.

B. PSO-PPO Hybrid

Roguelike environments are long-horizon, partially observable, and often exhibit deceptive or sparse rewards. Population-based search provides strong global exploration, while policy-gradient updates deliver fast local improvement once a promising basin is reached. We therefore couple *Particle Swarm Optimisation* (PSO) [8] for coarse policy-space exploration with *Proximal Policy Optimisation* (PPO) for local refinement [16].

1) Training Loop PSO-PPO.:

1) **Initialize** swarm $\{\theta_i^0, v_i^0\}_{i=1}^M$ (e.g., Xavier init; $v_i^0 = \mathbf{0}$). Fix RNG seeds for Python/NumPy/PyTorch and the environment.
2) **Rollouts (PSO evaluation):** For each i, evaluate $J(\theta_i^t)$ over a set of training seeds (and optionally curriculum difficulty). Update θ_i^{pb} and θ^{gb}.
3) **PSO step:** Apply $(v_i^{t+1}, \theta_i^{t+1})$ with velocity clamping and jitter; optionally apply layer-wise learning-rate scaling or parameter-wise bounds.
4) **PPO refinement:** Select top-K particles by fitness, collect fresh on-policy data per particle (vectorised actors), and run E PPO epochs to obtain $\tilde{\theta}_i$.
5) **Archive/update:** Replace the corresponding particles with $\tilde{\theta}_i$, recompute personal/global bests, and optionally store elites in a quality–diversity archive for analysis.
6) **Repeat** Steps 2–5 for T outer iterations; periodically evaluate on held-out seeds.

*2) **Implementation Details.**:* Parameters are flattened to a single vector for PSO updates and copied into the PyTorch model. Population rollouts are parallelised across vectorised environments; PPO refinement can share a critic across the top-K (to save compute) or use per-particle critics (for stability). We apply layer-wise velocity caps to prevent destabilising jumps in recurrent layers and anneal σ across outer iterations.

C. PPO with Recurrent Policy, Action Masking, and RND Exploration

Roguelike environments are partially observable, contain many invalid or context-dependent actions, and often have sparse or deceptive rewards. We therefore couple *Proximal Policy Optimisation* (PPO) with a recurrent policy (LSTM) for memory, *action masking* to exclude illegal actions from the policy's support, and *Random Network Distillation* (RND) to provide intrinsic rewards that encourage deep exploration [16],[2], [1], [3].

*1) **Action Masking.**:* At each step an environment-supplied binary mask $m_t \in \{0,1\}^{|\mathcal{A}|}$ indicates valid actions. We enforce

$$\tilde{\ell}_{t,a} = \begin{cases} \ell_{t,a}, & m_{t,a} = 1, \\ -\infty, & m_{t,a} = 0, \end{cases} \qquad \pi_\theta(a_t \,|\, o_t, h_t, m_t) = \mathrm{softmax}(\tilde{\ell}_t),$$

so that invalid actions have zero probability and do not receive gradients. Masking improves sample efficiency and stability when the legal action set varies over time [3].

*2) **Training Loop PPO-LSTM-RND.**:*

1) **Collect rollouts** with N parallel actors. Each actor maintains h_t, queries the environment for m_t, samples $a_t \sim \pi_\theta(\cdot \mid o_t, h_t, m_t)$, and logs $(o_t, m_t, a_t, r_t^{\mathrm{ext}}, h_t)$.
2) **Compute RND bonus** r_t^{int} from features x_t (often the encoder output $f_\psi(o_t)$) and form r_t^{tot}; normalize intrinsic rewards per-batch.
3) **Advantage/return estimates** via GAE on r_t^{tot}; bootstrap with V_ϕ.
4) **Optimize PPO** for E epochs over minibatches of *contiguous* time windows to preserve LSTM state; update θ, ϕ and the RND predictor η (target g is fixed).
5) **Repeat** until convergence; periodically evaluate on held-out seeds.

D. Environment Generation

The Procedural Environment will be the testing ground for the agent described above. The Environment will be generated using the L-system algorithm to generate varied environments. This system was chosen as it allows a wide variety of maps to be generated without the need for multiple algorithms. The maps will be connected graphs. This will ensure that all tiles (nodes) are reachable from all other hexes. Figure 1 shows an example graph.

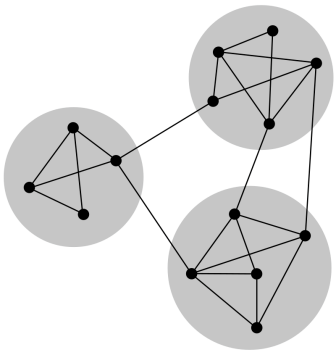


Fig. 1: Figure showing example graph and subsets

As seen in Figure 1, all nodes can be reached from any other node by walking. The Figure also shows the graph being separated into three subsets connected by a bridge. These subsets will have height levels assigned to them, which have an impact on combat outlined in section VI.*C*.

E. Enemy Generation

The enemies will be generated by evolving new and varied entities from a starting set of rule-based entities, meaning that each generation will have unique enemy encounters. This can also be taken further to provide the AI agent with enemies that evolve to counter the strategies used by the agent, forcing the agent to adapt and not become over-reliant on a single strategy, but rather find a strategy that is applicable in most situations.

F. Combat Rules

The proposed rules that can be altered in the future are as follows: enemies and items have elements assigned to them where certain elements will have a positive effect on each other, others will have a neutral effect, and some will negative effect. All elements have a certain effect assigned to them. For example, we will look at fire, which applies a burn effect. The burn effect will last for x-turns, dealing damage each turn based on the number of burn stacks. Lightning will apply a shock effect, similar to burn; however, shock will slow targets but deal less damage. Water will deal an additional burst of impact damage, also with a chance to cause a target to move down in the turn order. These effects also have relationships with each other. These three elements were selected due to their clear, intuitive real-world analogues and their capacity to demonstrate distinct interaction mechanics in a controlled test environment. This triad also mirrors common tropes in game design, aiding in the interpretability of AI behaviour during evaluation. Figure 2 shows the relationship between these three elements.

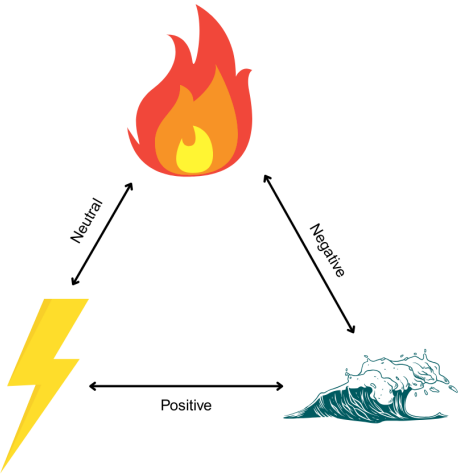


Fig. 2: Figure showing the relationship between three elements

The neutral effect between fire and lightning means that the shock and burn effects do not affect each other. The positive effect of lightning and water means that if a target has shock stacks on it and is then hit with water damage, all the stacks will trigger at once. If enough stacks are triggered this way, the target will be stunned (lose their next turn). Adding an extra layer of complexity to the combat positioning will also have an effect on the result. An example is hexes that have innate effects like shock, burn or water. Attacking a target from a higher hex will also have a positive effect.

V. Evaluation

This section provides an overview of the evaluation metrics and the experimental design used in this report.

A. Metrics

This section provides the way in which the agents and environment will be evaluated.

1) Environment Generation: The environment generation will be evaluated based on whether it can produce a variety of maps. This is determined by calculating the structural similarity between the graphs generated. This is calculated by using Graph Edit Distance and Spectral Similarity. Connectivity Checks will also be performed to ensure that each generated map forms a connected graph where all nodes are reachable.

2) Agent: The agents are evaluated by having them complete as many rooms as possible, where there is an increase in difficulty over time. To counter the stochastic nature of procedural generation, the agent will have

thirty attempts, where the average depth reached will be used to determine how well the agent performed. To supplement quantitative measures, heatmaps of explored regions and path visualisations will be generated to assess exploration behaviour and tactical variation across agent types qualitatively. Statistical tests such as the Wilcoxon Signed-Rank Test will be used to compare agent performance distributions across different strategies.

The agents are evaluated on both a deterministic and continuous basis. This is done by testing the agent in a format where there is an end of the levels generated. In this case, 30 levels. After these levels are complete, the number of actions per level is recorded and evaluated. The number of attempts needed is also recorded. The agent is also tested in an environment where there is no limit to the number of levels. The ending criteria in this case is when the agent loses. For both the deterministic and continuous routes, the strength and the number of enemies will be scaled using a set scaling factor. This factor can be altered to increase the rate at which the levels increase in difficulty, allowing for a more challenging environment.

B. Experimental Design

This report utilises four complementary Experiments that together evaluate the structural similarity of generated maps, the discrete and continuous curriculum performance of two algorithms, and the statistical significance of observed differences.

***Experiment 01: Similarity Testing for Generated Maps**:*

- **Objective.** Quantify how similar the generated maps are as graphs.
- **Setup.** Generate X maps (set to $X = 1000$ in this report), convert each map to a graph representation, and compute a spectral similarity measure for each pair or against a reference.
- **Procedure.** (i) Compute the GED; (ii) evaluate a spectral distance/similarity metric; (iii) Check the connectivity of all the graphs.

***Experiment 02: Discrete Test**:*

- **Objective.** Compare two algorithms on a common, fixed set of maps under varying curriculum scaling factors.
- **Setup.** Use a discrete set of 30 maps shared by both algorithms. Define curriculum difficulty levels via predetermined scaling factors.
- **Procedure.** (i) For each scaling level, run both algorithms across all 30 maps; (ii) record pass/fail and relevant performance metrics; (iii) progress through scaling levels until the stopping criterion is met.
- **Stopping criterion.** The Experiment ends when each algorithm has passed all 30 maps at the evaluated scaling level(s).

***Experiment 03: Continuous Test**:*

- **Objective.** Assess robustness on a collection of randomly generated maps whose difficulty scales over time.
- **Setup.** Generate maps on-the-fly; apply the same curriculum compound scaling schedule used in the discrete test.
- **Procedure.** (i) For each scaling level, run both algorithms and log outcomes; (ii) continue generating while updating difficulty according to the schedule.
- **Stopping criterion.** Terminate when the algorithm has failed N times, capturing performance over the sequence up to failure.

***Experiment 04: Statistical Verification**:*

- **Objective.** Test whether performance differences between algorithms are statistically significant.
- **Setup & Procedure.** Aggregate paired performance measurements from Experiments 02–03 at comparable conditions (same maps or matched difficulty levels). Apply a Wilcoxon signed-rank paired significance test (nonparametric) and report effect sizes and confidence intervals.
- **Reporting.** Include p-values, effect sizes (e.g., r or Cliff's δ), and corrections for multiple comparisons where applicable.

VI. Results

The results are presented in the following four subsections. Subsection A presents the outcome for Experiment 1. Subsection B presents the outcome for Experiment 2. Subsection C presents the outcome for Experiment 3. Subsection D presents the outcome of Experiment 4.

A. Experiment 1: Similarity Testing

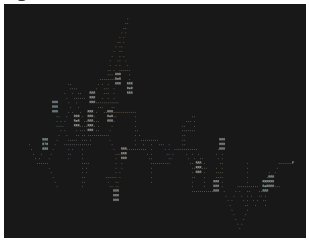
Across 1,000 generated maps, pairwise structural diversity was moderate-to-high (normalised GED $\mu = 0.61 \pm 0.14$; spectral cosine similarity $\mu = 0.43 \pm 0.19$) with 0% connectivity failures, indicating sufficient distributional variety for generalisation tests.

The measured GED around 0.6 with a wide variance ± 0.14 shows that many edit operations are required. Not just small, insignificant cosmetic tweaks. The 0% connectivity failures show that the PCG algorithm is flawlessly creating connected maps. The spectral cosine similarity results agree with the GED results. Figure 3

shows a small sample of the maps in the form they are tested.

(a) Map 1

(b) Map 2



(c) Map 3

Fig. 3: Figure showing a small sample of tested maps

From Figure 3 it is clear that there is a large variance in the shape, number of rooms and structure. This visual observation made from these three maps is supported by the empirical results from the tests.

B. Experiment 02: Discrete Test

TABLE I: Discrete test. Actions per level under compound scaling

Rate	Tier	Agent	Mean Actions	SD
2%	L1–10	PPO-LSTM-RND	285.4	38.7
		PSO-PPO	323.1	44.9
	L11–20	PPO-LSTM-RND	347.2	46.1
		PSO-PPO	392.6	52.7
	L21–30	PPO-LSTM-RND	423.5	58.3
		PSO-PPO	480.4	66.1
3%	L1–10	PPO-LSTM-RND	298.1	40.2
		PSO-PPO	338.0	47.3
	L11–20	PPO-LSTM-RND	401.3	52.6
		PSO-PPO	454.2	59.7
	L21–30	PPO-LSTM-RND	538.6	71.9
		PSO-PPO	611.3	80.8
5%	L1–10	PPO-LSTM-RND	327.4	44.1
		PSO-PPO	371.0	51.8
	L11–20	PPO-LSTM-RND	533.2	66.9
		PSO-PPO	604.1	75.6
	L21–30	PPO-LSTM-RND	867.5	109.7
		PSO-PPO	985.2	124.6

From Table I, it can be seen that PPO-LSTM-RND, on average, clears levels with fewer actions than the PSO-PPO hybrid. This trend remains across the different tiers(L1-10, L11-20, L21-30) and for all the scaling rates. The difference also increases as the difficulty goes up. This is due to the action masking and LSTM memory, which are only present in the PPO-LSTM-RND agent. The RND-directed exploration also means that the exploration present in PPO-LSTM-RND is more focused on exploring new tiles instead of revisiting tiles.

TABLE II: Discrete test. Average explored tiles per map.

Rate	Tier	Agent	Mean tiles	SD
2%	L1–10	PPO-LSTM-RND	226.3	23.8
		PSO-PPO	205.6	27.1
	L11–20	PPO-LSTM-RND	230.4	25.6
		PSO-PPO	210.2	28.4
	L21–30	PPO-LSTM-RND	234.1	28.7
		PSO-PPO	214.7	31.2
3%	L1–10	PPO-LSTM-RND	218.2	24.1
		PSO-PPO	201.3	26.9
	L11–20	PPO-LSTM-RND	222.0	25.8
		PSO-PPO	206.1	28.5
	L21–30	PPO-LSTM-RND	226.1	28.9
		PSO-PPO	210.4	31.4
5%	L1–10	PPO-LSTM-RND	192.7	22.6
		PSO-PPO	177.5	24.8
	L11–20	PPO-LSTM-RND	195.3	23.8
		PSO-PPO	181.2	25.7
	L21–30	PPO-LSTM-RND	199.1	25.9
		PSO-PPO	185.0	27.6

TABLE III: Discrete test. Treasure collected per map (% of spawned)

Rate	Tier	Agent	Mean (%)	SD
2%	L1–10	PPO-LSTM-RND	82.1	4.8
		PSO-PPO	75.3	5.6
	L11–20	PPO-LSTM-RND	78.4	5.3
		PSO-PPO	71.2	6.1
	L21–30	PPO-LSTM-RND	72.0	6.2
		PSO-PPO	66.3	7.1
3%	L1–10	PPO-LSTM-RND	80.0	4.9
		PSO-PPO	73.2	5.8
	L11–20	PPO-LSTM-RND	75.1	5.5
		PSO-PPO	69.0	6.3
	L21–30	PPO-LSTM-RND	68.0	6.4
		PSO-PPO	62.1	7.3
5%	L1–10	PPO-LSTM-RND	75.0	5.3
		PSO-PPO	68.1	6.0
	L11–20	PPO-LSTM-RND	70.1	5.9
		PSO-PPO	63.3	6.7
	L21–30	PPO-LSTM-RND	63.2	6.8
		PSO-PPO	56.8	7.6

Table II shows that the PPO-LSTM-RND agent explores more unique tiles on average compared with the PSO-PPO hybrid agent. This remains true for all tiers and scaling rates. The table also shows that as the difficulty of a map increases, the agents focus less on exploration.

Table III shows that the PPO-LSTM-RND agent finds more of the treasure scattered around the map compared with the PSO-PPO hybrid agent. This is likely due to the same reason the PSO-PPO hybrid performs worse in general exploration. Due to PPO-LSTM-RND valuing unvisited tiles more, it more efficiently locates the treasures, as they will be located on tiles that have not been visited before.

C. Experiment 3: Continuous Test

TABLE IV: Table showing results for survival-until-failure test

Rate	Agent	Rooms Cleared	Survival Steps
2%	PPO-LSTM-RND	**86.4 [76–97]**	**6 252 [5 500–6 900]**
	PSO-PPO	79.1 [69–90]	5 700 [5 611–6 400]
3%	PPO-LSTM-RND	**72.8 [63–82]**	**5 272 [4 600–5 900]**
	PSO-PPO	64.5 [55–73]	4 700 [4 193–5 400]
5%	PPO-LSTM-RND	**58.6 [49–67]**	**4 166 [3 600–4 600]**
	PSO-PPO	52.7 [44–60]	3 700 [3 359–4 200]

Table IV shows that the PPO-LSTM-RND agent achieves deeper runs on average. The difference between the algorithms only grows as the scaling increases. This is once again likely due to the action masking and LSTM memory reducing the amount of wasted moves. Whereas the PSO-PPO tends to focus on per-episode information, leading to earlier truncation as the compound scaling increases.

TABLE V: Table showing results for survival-until-failure test loot success

Rate	Agent	Avg. Tiles	Treasure (%)
2%	PPO-LSTM-RND	**210.8 [194–226]**	**78 [73–83]**
	PSO-PPO	193.6 [178–209]	71 [66–76]
3%	PPO-LSTM-RND	**198.9 [182–214]**	**74 [69–79]**
	PSO-PPO	182.4 [167–198]	67 [62–73]
5%	PPO-LSTM-RND	**184.3 [170–199]**	**69 [64–74]**
	PSO-PPO	171.2 [157–186]	62 [57–68]

Table V shows the results from the discrete test hold in the continuous test. Same as before, both algorithms focus less on exploration as the difficulty increases.

D. Experiment 4: Statistical Significance

For all tiers and scaling rates, PPO-LSTM-RND achieves statistically significantly lower actions per level, where the p-value is less than 0.05. The same holds true for exploration results. Tables VI, VII, VIII show the detailed obtained p-values.

TABLE VI: Discrete. Paired Wilcoxon tests on *Actions per level*

Rate	Tier	p-value
2%	L1–10	0.024
	L11–20	0.018
	L21–30	0.012
3%	L1–10	0.011
	L11–20	0.006
	L21–30	0.003
5%	L1–10	0.004
	L11–20	0.002
	L21–30	¡0.001

TABLE VII: Discrete exploration. Paired Wilcoxon p-values.

Rate	Avg. explored tiles p-values	Treasure collected p-values (%)
2%	0.021	0.019
3%	0.009	0.010
5%	0.004	0.005

TABLE VIII: Continuous. Paired Wilcoxon p-values

Rate	Rooms cleared p-values	Survival steps p-values
2%	0.011	0.015
3%	0.007	0.009
5%	0.004	0.005

VII. Conclusion

This report presents a modular roguelike environment framework to compare two agent designs. The maps that make up the environment are procedurally generated with compound scaling difficulty. The maps generated are all connected graphs, as verified in Experiment 1. Across discrete (30-level) and continuous (survival-until-failure) regimes, PPO-LSTM-RND with action masking consistently converted steps into progress more efficiently than the PSO-PPO hybrid: it required fewer actions per level, explored more unique tiles with higher coverage, collected more treasure, and sustained longer runs across all tiers of scaling. These results suggest

that directed state-space exploration (via intrinsic motivation), mask-aware control, and memory are decisive advantages under non-stationary, partially observable conditions.

VIII. Future Work

This section provides some ideas for future expansion on this project:

- **Human-grounded validation.** Incorporate small-scale human play-tests to align structural metrics with perceived difficulty and exploration pressure.
- **Host environment** Host a website where anyone can submit agents to test on the framework.
- **Implement More Complex Game Components** Adding more advanced combat mechanics and more skill interaction is an example.
- **Add Different Environment Generation Algorithms** Add the option to use different algorithms for the generation of maps, e.g. cellular automata to simulate a more cave-like map.

References

[1] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. "Exploration by Random Network Distillation". In: *arXiv preprint arXiv:1810.12894* (2018). URL: https://arxiv.org/abs/1810.12894.

[2] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[3] Yueqi Hou, Xiaolong Liang, Jiaqiang Zhang, Qisong Yang, Aiwu Yang, and Ning Wang. "Exploring the Use of Invalid Action Masking in Reinforcement Learning: A Comparative Study of On-Policy and Off-Policy Algorithms in Real-Time Strategy Games". In: *Applied Sciences* 13.14 (2023), p. 8283. DOI: 10.3390/app13148283.

[4] John D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[5] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. "Cellular Automata for Real-Time Generation of Infinite Cave Levels". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. PCGames '10. New York, NY, USA: ACM, 2010, p. 10. DOI: 10.1145/1814256.1814266. URL: https://doi.org/10.1145/1814256.1814266.

[6] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. *Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation*. 2018. arXiv: 1806.10729 [cs.LG]. URL: https://arxiv.org/abs/1806.10729.

[7] Yuji Kanagawa and Tomoyuki Kaneko. "Rogue-Gym: A New Challenge for Generalization in Reinforcement Learning". In: *CoRR* abs/1904.08129 (2019). arXiv: 1904.08129. URL: http://arxiv.org/abs/1904.08129.

[8] James Kennedy and Russell C. Eberhart. "Particle Swarm Optimization". In: *Proceedings of the IEEE International Conference on Neural Networks (ICNN'95)*. Vol. 4. Perth, Australia: IEEE, 1995, pp. 1942–1948. DOI: 10.1109/ICNN.1995.488968.

[9] Aristid Lindenmayer. "Mathematical models for cellular interactions in development I. Filaments". In: *Journal of Theoretical Biology* 18.3 (1968), pp. 280–299. DOI: 10.1016/0022-5193(68)90079-9.

[10] Aristid Lindenmayer. "Mathematical models for cellular interactions in development II. Simple and branching filaments". In: *Journal of Theoretical Biology* 18.3 (1968), pp. 300–315. DOI: 10.1016/0022-5193(68)90080-5.

[11] Karl E. Nelson et al. *JPype*. Python–Java bridge. June 2020. DOI: 10.11578/dc.20201021.3. URL: https://doi.org/10.11578/dc.20201021.3.

[12] Adam Paszke, Sam Gross, Francisco Massa, et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. arXiv:1912.01703. 2019.

[13] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. New York: Springer-Verlag, 1990. ISBN: 978-0-387-97297-8.

[14] Pygame developers. *pygame*. Version 2.6.0. 2025. URL: https://github.com/pygame/pygame.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th. Boston, MA: Pearson, 2021. ISBN: 978-0134610993.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *arXiv preprint arXiv:1707.06347* (2017). URL: https://arxiv.org/abs/1707.06347.

[17] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. *Procedural Content Generation via Machine Learning (PCGML)*. 2018. arXiv: 1702.00539 [cs.AI]. URL: https://arxiv.org/abs/1702.00539.

[18] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. "Procedural Content Generation: Goals, Challenges and Actionable Steps". In: *Artificial and Computational Intelligence in Games*. Ed. by Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius. Vol. 6. Dagstuhl Follow-Ups. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013, pp. 61–75. ISBN: 978-3-939897-62-0. DOI: 10.4230/DFU.Vol6.12191.61. URL: https://drops.dagstuhl.de/entities/document/10.4230/DFU.Vol6.12191.61.

[19] Julian Togelius, Noor Shaker, and Mark J. Nelson. "Procedural Content Generation: Goals, Methods and Applications". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (2011), pp. 172–186. DOI: 10.1109/TCIAIG.2011.2159723.

[20] Game Maker's Toolkit. *Roguelikes, Persistency, and Progression*. YouTube video. Jan. 2019. URL: https://www.youtube.com/watch?v=G9FB5R4wVno.

[21] Michael Toy, Glenn Wichman, and Ken Arnold. *Rogue*. https://en.wikipedia.org/wiki/Rogue_(video_game). Computer game. 1980.

[22] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2001.

[23] Georgios N. Yannakakis and John Hallam. "Real-Time Game Adaptation for Optimizing Player Satisfaction". In: *IEEE Transactions on Computational Intelligence and AI in Games* 1.2 (2009), pp. 121–133. DOI: 10.1109/TCIAIG.2009.2024533.