Scaling, Control and Generalization in Reinforcement Learning Level Generators

Sam Earle
New York University
Game Innovation Lab
Brooklyn, NY
sam.earle@nyu.edu

Zehua Jiang
New York University
Game Innovation Lab
Brooklyn, NY
zehua.jiang@nyu.edu

Julian Togelius
New York University
Game Innovation Lab
Brooklyn, NY
julian@togelius.com

2024 IEEE Conference on Games (CoG) | 979-8-3503-5067-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/CoG60054.2024.10645598

***Abstract*—Procedural Content Generation via Reinforcement Learning (PCGRL) has been introduced as a means by which controllable designer agents can be trained based only on a set of computable metrics acting as a proxy for the level's quality and key characteristics. While PCGRL offers a unique set of affordances for game designers, it is constrained by the compute-intensive process of training RL agents, and has so far been limited to generating relatively small levels. To address this issue of scale, we implement several PCGRL environments in Jax so that all aspects of learning and simulation happen in parallel on the GPU, resulting in faster environment simulation; removing the CPU-GPU transfer of information bottleneck during RL training; and ultimately resulting in significantly improved training speed. We replicate several key results from prior works in this new framework, letting models train for much longer than previously studied, and evaluating their behavior after 1 billion timesteps. Aiming for greater control for human designers, we introduce randomized level sizes and frozen "pinpoints" of pivotal game tiles as further ways of countering overfitting. To test the generalization ability of learned generators, we evaluate models on large, out-of-distribution map sizes, and find that partial observation sizes learn more robust design strategies.**

***Index Terms*—procedural content generation, reinforcement learning**

I. Introduction

In procedural content generation via reinforcement learning (PCGRL), the process of iterative game design is frames as a markov decision process, and reinforcement learning (RL) agents are trained to generate game content. Instead of learning to play a game by taking actions, observing states, and getting rewards, these agents learn to generate (parts of) a game by taking actions, observing states, and getting rewards. The actions edit the content artifact, and the reward is based on the quality of the artifact that is being created.

The advantage of PCGRL is that you can use it to create not just game content, but game content *generators*. Compared to search-based approaches, this means that almost all the compute is front-loaded; first you train the generator, then inference is fast and cheap. This makes it suitable for runtime use in games. Compared to supervised or self-supervised learning, PCGRL don't need any existing content to train on. This makes it suitable for use for games where any content has yet to be produced.

Despite these considerable potential advantages, PCGRL has only limited uptake since it was first proposed in [1]. This could be due to the difficulty of designing good reward functions, the tendency to overfit to single solutions, the long training time, and the problems with scaling to produce larger-size levels and other content. In this paper, we propose and evaluate several modifications to the basic PCGRL formulation aimed at rectifying some of these issues.

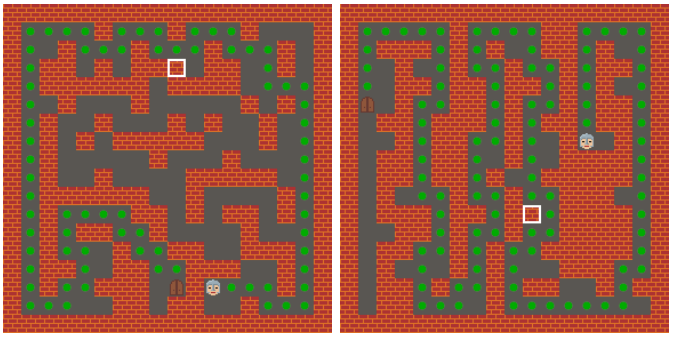
Two novel elements we propose are randomizing level size during training, and pinpointing locations of key elements. Both of these interventions function to limit overfitting by enforcing closed-loop policies, in other words, the agent must take its observations into account and cannot rely on rote-learning parts of levels. These add new degrees of controllability in addition on top of conditioning on high-level features introduced in [2].

We also examine the effects of systematically changing the size of the agent's observation space. In the original PCGRL formulation, the observation window typically covers the whole level. From reinforcement learning experiments in various domains, we know that observation and structure can have large effects on overfitting and scalability. We hypothesize that the same is true for PCGRL, and that we can improve generalization and scalability by choosing adequate observation windows. This hypothesis is largely confirmed by our experiments. We find that smaller observation windows always increase generalization to new level sizes. On a task involving pinpoints and randomized map shapes during training, these more local models additionally perform comparably or better in-distribution.

To offset the high computational cost of training content-generating agents, we reimplement the standard PCGRL library in jax [3], a framework that allows a high degree of parallelization using the GPU to simulate the environment, resulting for $15\times$ speedups during training, making it feasible to experiment with longer training times.

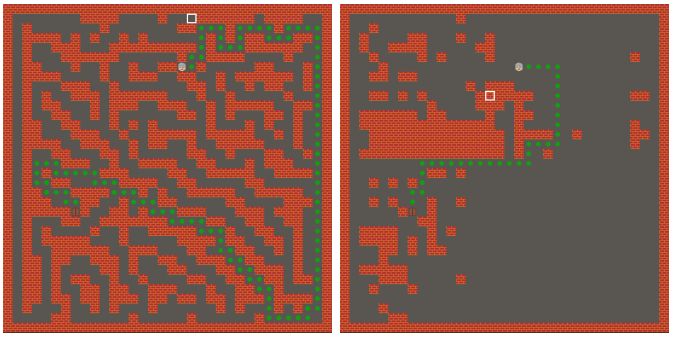
In sum, our contributions are as follows

- We re-implement the PCGRL code-base in jax, making it practical to scale PCGRL to larger and more complex domains.
- We add new features—variable map shapes and varied (frozen) placement of pivotal "pinpoint" tiles—to make



(a) 8×8 observations (b) 31×31 (global) observations

(c) Evaluation on in-distribution 16×16 maps.



(d) 8×8 observations (e) 31×31 (global) observations

(f) Evaluation on out-of-distribution 32×32 maps.

Fig. 1: Evaluation in the MAZE domain with pinpoints (randomly fixed player and door tiles). While models with large global observations are better on small 16×16 in-distribution maps, models with smaller local observations learn scalable patterns that generalize better to larger 32×32 maps.

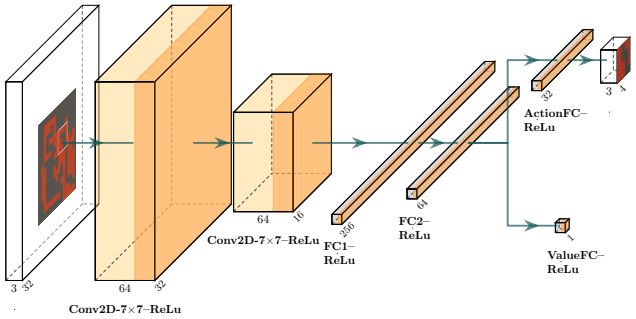
the task of level generation more complex, and to make the result generators more controllable.

- We conduct a thorough investigation of the effects of partial observations, finding that partial observations are more successful in generalizing to large map sizes unseen during training.

II. BACKGROUND

Video games featuring some form of content generation, often level generation, have existed since the early 1980s. Beneath Apple Manor, Rogue, and Elite were the early poster children of early game content generation. PCG has become a mainstay in modern games, with titles such as No Man's Sky, Hades, and the Civilization series that rely heavily on some form of runtime content generation.

Academic research in content generation dates back to the early 2000s [4]. Much of the early research focused on search-based approaches [5]. Search-based approaches are very versatile, but the computational demands at generation time can be high, making such approaches harder to use for runtime generation. Constraint satisfaction approaches [6; 7]



(a) **CONV model.** Both action and value branches process a single local or global 2D observation with convolutional and fully connected layers.

Fig. 2: **Model architectures.**

were also given considerable attention. While constraint satisfaction approaches can be very powerful, it imposes particular constraints on the shape of the content.

Supervised and self-supervised approaches to PCG started being explored seriously at the dawn of the deep learning era [8; 9; 10]. A variety of these machine learning methods have been applied to game content generation, including Generative Adversarial Networks [11], LSTM networks [12], and Markov models [13]. However, these methods generally have large requirements on training data. This begs the question that has been called the fundamental tension of PCGML: *if these methods only work well with enough existing content, why would you need to generate it?* [14].

Reinforcement learning approaches to PCG are more recent, first proposed by [15] and [1]. There are significant advantages to PCGRL over existing approaches:

1) No training data is necessary
2) Generators are very fast during inference time
3) The generator is iterative, allowing mixed-initiative solutions [16]

With these advantages comes a unique set of considerations. In principle, the same kind of evaluations used as fitness functions for search-based PCG can be used as reward functions in PCGRL. However, due to longer training times, a computationally lightweight reward function is needed in settings with dense reward. There is also a tendency to mode collapse via overfitting, which can be counteracted via limiting the number of changes the model can make, or introducing conditional inputs [2]. Because of these constraints, scaling PCGRL to larger-sized levels or artefacts has proven a challenge [17].

One type of modification to the basic PCGRL formula explored here concerns the size and shape of the observation. This draws on earlier results showing that limiting the observation size and aligning it properly can greatly help with generalization [18].

	n. envs	1	10	50	100	200	400	600
CPU	binary	189	420	520	524	525	529	522
	dungeon	206	416	505	511	509	511	496
jax	binary	256	1,256	1,098	3,993	6,076	7,814	8,943
	maze	471	2,099	6,159	8,039	9,319	9,456	10,961
	dungeon	292	1,463	4,042	5,375	6,851	7,502	8,798

TABLE I: Environment steps per second on various domains in the CPU and jax implementations of [2] during training, on 10 CPUs or an RTX-8000 respectively, under varying numbers of parallelized environments.

n. envs	50	100	200	400	600
binary	6846.73	11979.56	21478.48	37072.85	46498.75
maze	6867.30	12325.76	21902.67	37745.91	47323.79
dungeon	6923.89	12349.95	21882.25	37770.10	46839.11

TABLE II: Environment steps per second on various domains in the jax implementation of PCGRL while taking random actions, on an RTX-8000, under varying numbers of parallelized environments. Frames per-second exceed 45k, despite the relative complexity of the pathfinding operations required to compute reward in these domains.

III. Methods

A. Training

To train RL level generators, we use Proximal Policy Optimization (PPO) [19] with the same reward function and neural network as in [1; 2], using the "narrow" representation of observations and actions. In each episode, the model is rewarded by minimizing the loss value between current state and target state (where the user or a training curriculum can vary target solution path length or nearest enemy). The agent observes an egocentric patch of the level (in which the board may be padded to allow for global observations), and may change the state of its current tile, then is moved to an adjacent tile in an iterative scan of the map.

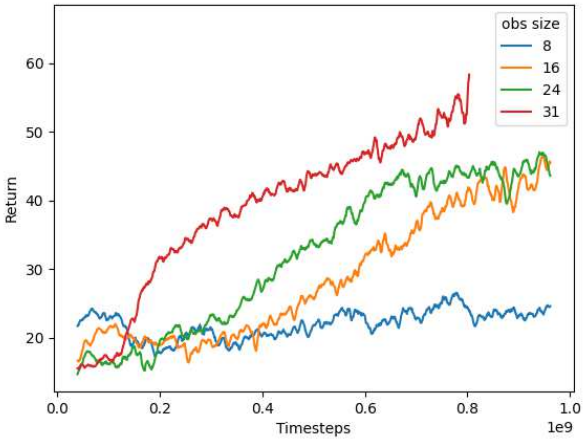
B. Task

We extend PCGRL [1], in which level design is framed as a reinforcement learning task. This task is decomposed into a "problem"—the level design task at hand—and a "representation"—the interface via which the agent edits the level. In this paper, we adapt the *narrow* representation to support new features, and use the *binary maze* and *dungeon* problems and as toy tasks with which to verify our proposed method. We initialize the map with the elements of the tasks using the weighted uniform distribution as in [1], or simply leave it empty, depending on the configuration of the environment. At each timestep, we compute the metrics of interest (i.e. the diameter and number of connected empty regions) if the agent has made any modification to the map.

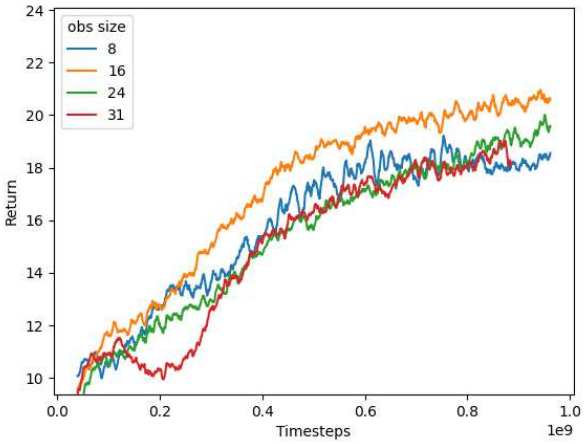
a) Binary domain: The agent's goal is to create a maze with maximum diameter (i.e. the longest shortest path between any two points in the maze). There are only two types of tile in the maze: wall and air. We approximate the diameter by applying Dijkstra's algorithm twice[1] and the number of connected components using a flood fill algorithm.

b) Maze domain: In this problem, the RL agent needs to use "wall" and "air" tiles to create a traversable maze from one "player" tile to "door" tile in the game map. The generated maze should just have one connected component and the solution should be maximized.

c) Dungeon domain: Expanding on the MAZE domain, we consider a task in where the agent's goal is to create a playable dungeon game level. The generated dungeon should



(a) Fixed map shapes during training.



(b) Randomized per-episode map shapes during training.

Fig. 3: Reward curve of the CONV model on the MAZE domain with pinpoints (randomly frozen player and door). On a more challenging task involving randomized per-episode map shapes, the performance gap between models with global and partial observations shrinks.

[1]First, we select a random empty tile x in the maze. We apply Dijkstra's algorithm to find the longest shortest path of which x is an endpoint. We then take y, the other endpoint of this shortest path, and apply Dijkstra's algorithm starting from y to find the longest shortest path of which y is an endpoint, under the assumption that y is an endpoint of the diameter.

have a maximally long shortest path from player, to key, to door. There are 6 types of tile in dungeon problem: wall, air, enemy, key, door, and player. There can be 2 to 5 enemy tiles, and only one key, door, and player tile. Additionally, the distance between the player and the nearest enemy should not be less than 4 tiles. We use Dijkstra's algorithm to find the shortest player-key-door path.

d) Pinpoint tiles: We add additional complexity to our tasks, modifying the MAZE and DUNGEON domains so that important tiles can be frozen in arbitrary positions on the board. This freezing can be done programmatically during training, or at inference time via a human designer. These fixed-position tiles provide a more controllable way to prevent agents fromoverfitting to a single optimal output, and allow the problem to become open-looped.

e) Randomized map shapes: To train agents that scale to larger and more variable map shapes w.r.t those seen during training, we expose the agent to a variety of map shapes throughout training. So, when training on a 16×16 map (as in prior PCGRL work) we randomly sample a rectangular map shape from a uniform distribution. The dimensions of this shape are bounded by 3×3 and 16×16. Another alternative might be to progressively train on maps of increasing size. Although intuitive, this approach introces two failrue modes. First, it risks catastrophic forgetting of smaller map sizes seen earlier during training. Conversely, it may incentivise the agent to learn faulty representations on smaller maps that do not transfer to larger ones.

C. Jax implementation

We use the jax python library [3] to implement our PCGRL environments and training algorithm. Jax exposes a wide variety of tensor-based operations to the user, mirroring much of the functionality of numpy and/or libraries like pytorch or tensorflow, compiling operations to XLA just-in-time. Provided that the size of tensors is fixed at compile-time, and with some limitations on logical operations like `if` conditions, Jax can improve runtime efficiency by "fusing" lower-level operations. Our Jax implementation of PCGRL builds on gymnax [20] and the PureJaxRL code base [21], which implements a number of simple embodied game-playing environments in Jax.

To pathfind in jax, we can flood activation out to adjacent traversible tiles in parallel across the board using convolutional kernels. Similar logic can be used to compute the number of regions.

IV. Results

a) Speed Comparison: In Table I we calculate the FPS during training of the prior CPU implementation (splitting environments across 11 cores), and jax-pcgrl respectively. We find that the jax implementation leads to speedups of over $15\times$ relative to the CPU implementation. While the CPU implementation plateaus as the number of environments reaches between 50-100, the FPS of the jax version continues to increase significantly up to at least 600 workers.

In Table II, we calculate the speed of our jax implementations of the BINARY MAZE and DUNGEON domains in frames per second given actions from a random agent, achieving 45k FPS.

b) Observation size: In the following experiments, we train the agents in a fixed size map setting and evaluate them on different map shapes. From these thorough experiment configurations, we can answer the question about how observation size influence the model performance on in-distribution and out-distribution map and the how well the agents generalize.

In Table III we evaluate the performance of the CONV model on the DUNGEON task. We evaluate different obseravtion input of the model on varying maximum random map widths and random per-episode map shapes. Performance of models with smaller observation windows is slightly weaker than models with full observations on in-distribution maximum map widths. But smaller observations lead to better generalization to larger maximum map widths, despite these models having fewer parameters then their fully-observing counterparts.

In Table IV, we evaluate the CONV model on the DUNGEON domain with random target path length for learning controllability. Looking at evaluation scenarios without randomized per-episode map shapes, we see that on in-distribution 16×16 maps, global observations perform the best. On larger maps, however, more local partial observations generalize better, despite the fact that models with smaller observation windows have significantly fewer parameters. When evaluating with randomized per-episode map shapes, however, models of all observation sizes generalize comparably.

We repeat this experiment in Table V, where we systematically add hidden nodes to each layer of each network until it has almost as many (but no more) learnable parameters than the model with full observations—effectively separating the effect of partial observations from model size. Here, we see that models with smaller observation size but comparable numbers of parameters tend to outperform models with larger observations across all maximum map sizes.

In Figure 5, we plot the reward curves of the experiments in Table IV and Table V during training, and similarly observe that the addition of learnable parameters to models with smaller observation sizes improves their performance.

c) Randomized map shapes during training: In Table VI, we examine the performance of the CONV model on the MAZE domain, with pinpoints (randomized fix placement of player and door tiles). In these experiments, we show how randomizing the map shape during training will affect the model performance of generalization under different observation size.

First, we look at evaluation of models on fixed square map sizes. In this setting, models exposed to similarly fixed square map shapes during training tend to outperform those trained on variable per-episode map shapes. Next, we evaluate models while randomly sampling shapes within these maximum sizes on each evaluation episode (right side of the table). Models trained without randomized map shapes are broadly unable to adapt to variable map shapes during evaluation—with the only

	mean ep reward							
rand. map shape	False				True			
eval map width	8	16	24	32	8	16	24	32
obs size								
15	20.43 ± 0.94	281.57 ± 74.31	**282.40 ± 1.95**	**497.95 ± 6.65**	6.19 ± 0.33	38.48 ± 0.38	96.22 ± 2.02	**140.16 ± 4.83**
20	22.82 ± 0.39	193.72 ± 21.75	272.26 ± 11.14	486.20 ± 20.73	7.06 ± 0.62	37.56 ± 1.89	**97.18 ± 1.50**	135.03 ± 2.00
25	**24.89 ± 2.22**	**338.47 ± 6.41**	276.06 ± 13.81	474.13 ± 33.43	**8.64 ± 1.70**	**39.44 ± 3.18**	96.55 ± 0.76	136.51 ± 1.12
31	18.04 ± 0.66	269.86 ± 29.98	273.20 ± 4.02	492.98 ± 6.93	7.51 ± 1.08	30.35 ± 3.28	91.37 ± 2.24	132.96 ± 5.04

TABLE III: Performance on the DUNGEON domain, of models with varying observation size. Models are trained on 16×16 maps of fixed shape during training, and evaluated on larger and variable-shaped maps. Smaller observation windows lead to better generalization on these tasks.

	mean ep reward							
rand. map shape	False				True			
eval map width	8	16	24	32	8	16	24	32
obs size								
3	**23.77 ± 2.21**	175.03 ± 14.62	**318.68 ± 24.37**	**548.01 ± 36.50**	7.70 ± 1.56	35.65 ± 2.71	99.90 ± 4.11	138.22 ± 3.69
5	22.66 ± 1.24	175.48 ± 2.56	288.45 ± 0.68	514.73 ± 2.17	8.40 ± 1.46	**37.56 ± 1.74**	**100.57 ± 5.48**	139.29 ± 3.46
8	19.69 ± 0.71	140.76 ± 2.66	285.70 ± 0.67	509.28 ± 2.80	7.23 ± 0.88	36.82 ± 1.16	99.73 ± 1.72	**145.10 ± 4.09**
16	18.53 ± 2.14	183.15 ± 13.76	282.80 ± 4.51	490.29 ± 6.66	5.34 ± 0.88	36.46 ± 1.59	98.90 ± 3.11	136.70 ± 1.52
31	20.40 ± 0.55	**184.12 ± 28.38**	273.73 ± 0.69	492.11 ± 1.62	**8.60 ± 0.85**	35.48 ± 0.38	93.54 ± 2.61	136.60 ± 3.06

TABLE IV: Performance on the DUNGEON domain, with controllable path length, of the CONV model with varying observation size. Smaller observation windows often lead to better generalization.

	mean ep reward							
rand. map shape	False				True			
eval map width	8	16	24	32	8	16	24	32
obs size hid dims								
3	**37.21 ± 9.46**	188.29 ± 8.83	**372.66 ± 3.49**	**582.95 ± 30.90**	**10.89 ± 2.20**	39.89 ± 1.92	**103.75 ± 0.95**	141.59 ± 2.10
5	24.26 ± 0.89	193.17 ± 17.19	295.83 ± 6.47	517.08 ± 0.72	9.87 ± 0.60	41.39 ± 3.70	103.48 ± 5.41	**145.19 ± 5.50**
8	27.00 ± 0.78	181.41 ± 15.48	316.44 ± 17.98	520.65 ± 6.83	8.53 ± 0.17	38.99 ± 2.88	103.19 ± 1.98	139.42 ± 0.81
16	19.25 ± 0.72	182.88 ± 8.26	265.95 ± 11.06	451.51 ± 24.09	5.80 ± 0.14	**44.19 ± 3.45**	100.01 ± 6.53	135.75 ± 6.24
31	17.43 ± 0.77	**206.98 ± 21.27**	276.81 ± 5.06	490.94 ± 15.53	5.03 ± 0.81	31.03 ± 1.30	92.21 ± 3.50	133.53 ± 1.93

TABLE V: Performance on the DUNGEON domain, with controllable path length, of the CONV model with varying observation size—while fixing the number of learnable model parameters. Relative to Table IV, increasing the number of parameters in partially-observing models leads to improvements on certain out-of-distribution scenarios (though not on randomized map shapes).

		mean ep reward							
	rand. map shape	False				True			
	eval map width	8	16	24	32	8	16	24	32
rand. map shape	obs size								
False	8	**7.18 ± 0.97**	22.45 ± 6.02	**44.57 ± 4.79**	**59.07 ± 3.78**	**1.23 ± 0.59**	**4.82 ± 1.59**	**9.57 ± 4.40**	**6.10 ± 7.43**
	16	3.93 ± 4.43	48.17 ± 2.70	39.53 ± 9.98	20.08 ± 8.51	0.27 ± 0.48	-1.28 ± 4.50	-3.15 ± 2.87	-8.14 ± 3.09
	24	2.31 ± 1.42	41.87 ± 1.70	18.55 ± 7.21	0.43 ± 8.78	0.87 ± 0.64	1.07 ± 1.08	1.11 ± 0.92	0.79 ± 0.46
	31	3.19 ± 0.95	**51.37 ± 7.41**	-10.18 ± 5.56	-14.73 ± 9.70	0.93 ± 0.03	-2.33 ± 3.87	-5.97 ± 5.22	-6.71 ± 6.92
True	8	**9.24 ± 0.30**	**22.13 ± 3.08**	**29.88 ± 9.93**	**30.48 ± 9.47**	**6.61 ± 0.16**	20.09 ± 2.70	**26.11 ± 4.08**	**25.12 ± 1.85**
	16	5.36 ± 1.36	13.38 ± 2.82	0.42 ± 4.86	-9.63 ± 6.02	5.93 ± 0.39	**21.75 ± 1.76**	16.57 ± 2.79	16.87 ± 4.97
	24	5.05 ± 1.13	-3.97 ± 4.97	-19.45 ± 5.74	-31.21 ± 16.32	5.08 ± 0.76	18.31 ± 1.93	5.67 ± 3.70	-0.84 ± 3.65
	31	4.25 ± 1.43	-1.48 ± 1.82	-9.69 ± 7.61	-16.08 ± 11.71	3.76 ± 0.21	18.69 ± 0.55	3.82 ± 1.73	-0.68 ± 2.19

TABLE VI: Performance on the pinpointed MAZE domain, of the CONV model, with varying observation size and with or without exposure to randomized map shapes during training. When map shapes are randomized during training, the in-distribution performance gap between local and global models closes.

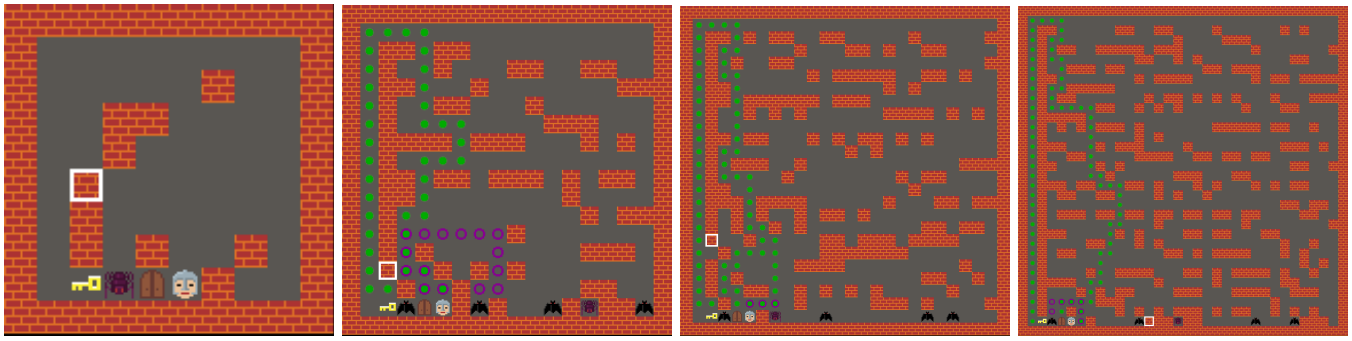
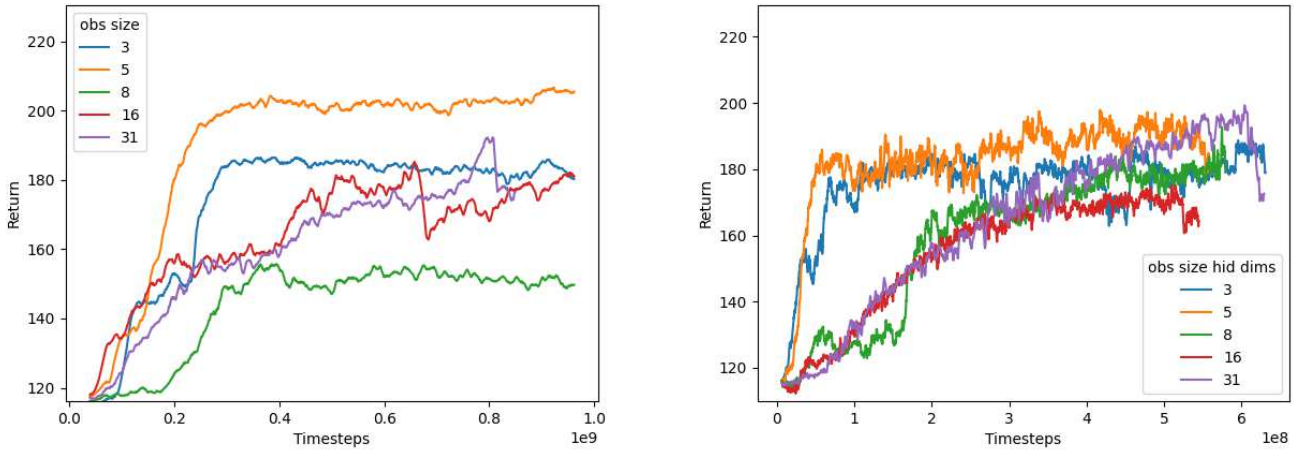


Fig. 4: On the DUNGEON domain with controllable path length the CONV model with 3×3 observation generalizes a design pattern to larger map sizes.



(a) Models with smaller observation windows have fewer parameters.

(b) Models with smaller observation windows have greater hidden dimensions and a similar number of learnable parameters.

Fig. 5: Effect of different observation sizes on reward curves during training of the CONV model on the DUNGEON domain with control targets.

consistent positive reward in this setting coming from models with the smallest observation window (8×8). Of the models trained on variable map shapes, smaller observation windows generally outperform larger ones.

In Figure 3, we plot reward curves of these models. When map shapes are fixed between episodes (Figure 3a), models with full observations outperform those with local observations. But when map shapes are randomized over each episode (Figure 3b), smaller observation windows lead to comparable or better performance than those with full observations. This trend is also replicated at evaluation time in Table VI, and can be observed qualitatively in Figure 1

V. DISCUSSION

To the best of our knowledge, no prior jax RL environments involving path-finding, which could also be a useful addition to player environments (e.g. to simulate enemy navigation in [22]). If the speedups Table II of our jax implementations of PCGRL environments are less than those of other, still simpler environments, this might come from the added complexity of our path-finding implementation.

Table VI shows us that in general, smaller observation windows lead to higher performance on out-of-distribution settings, either for larger scale or per-episode map shapes. We contend that this is likely a result of overfitting under global observation: models that are accustomed to seeing the padding of unique "border" tiles surrounding the effective map region are disrupted when, during evaluation on larger maps, these tiles are suddenly not present in their egocentric observations. These models are likely using the placement of these border tiles to infer global coordinates, allowing it to consistently construct one optimal global level (or a set of global levels, when dealing with randomized map shapes, controllable path-length metrics, or frozen "pinpoint" tiles). With such global coordinates at hand, these levels can theoretically be generated in one shot (or one scan over the board).

By restricting the observation space, on the other hand, models could only infer global coordinates by editing the

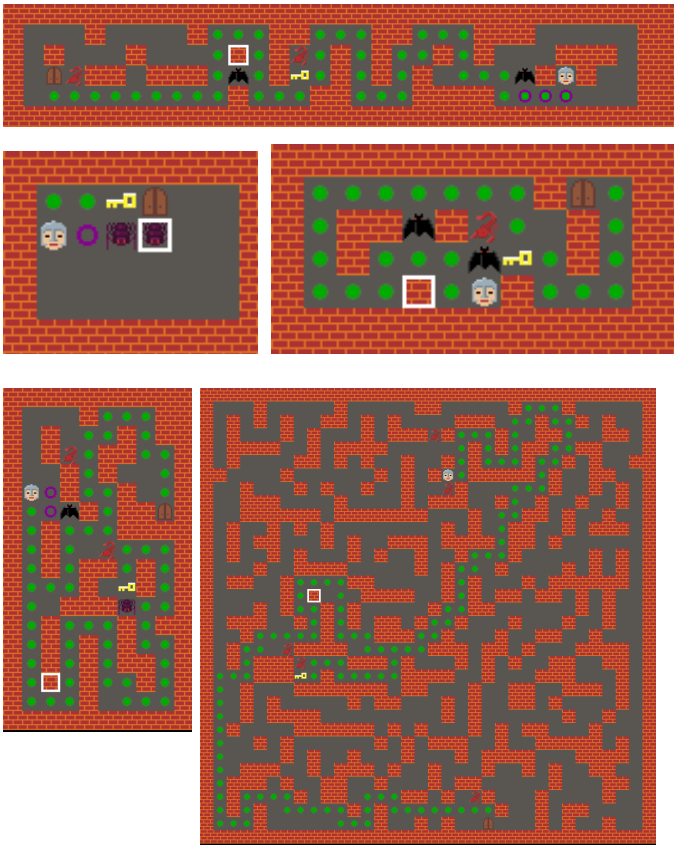


Fig. 6: A model taking local 8×8 observations generalizes a design strategy across varied map shapes.

entire board multiple times, communicating relative coordinates via patterns that cascade across the map in an iterative way. The fact that these local-observation models perform better out of distribution suggests that such an approach to iteratively passing local information across the board leads to more general representations and strategies for designing good levels. In other words, these constrained models are less likely to memorize *what* one or a set of optimal levels should look like, and instead may learn general strategies for *how* to improve or modify maps along certain axes.

Meanwhile, models trained on fixed-size square maps are not able to adapt well to per-episode variation of map shape. Models with full observations exhibit particularly pronounced failure in this case, and our reasoning would expect that a model that has observed only square border shapes is disrupted when it observes rectangular shapes during evaluation. But models with local observations are also thrown off by variable per-episode map shapes, suggesting that their strategies for iteratively transmitting local information are not robust to non-square map shapes. Conversely, maps trained on variable per-episode map shapes do not quite attain the performance of models trained strictly on fixed-size square maps—even on out-of-distribution sizes. We expect that this is merely a result of insufficient training time given the larger task distribution on which these models are trained, and that further training would allow them to better cover this distribution with good performance.

The reward curves in Figure 3, along with the in-distribution columns of Table VI, reveal that when per-episode map shapes are randomized, models taking full observations seem to lose the advantage they have when map shapes are fixed. This would seem to suggest that knowing the overall shape of the map is actually a disadvantage, even on in-distribution tasks, when this distribution is diverse enough. In other words, we hypothesize that models that are forced to learn general level-editing strategies, adaptive to a range of possible map shapes, arrive more quickly at optimal performance on the set of training map shapes, while models that have access to this information are effectively distract, and drawn away from more general and robust strategies. Or, when the training distribution is wide enough, limiting a model's direct access to information about precisely which training task it is in at a given moment can render it more effective, because this model is forced to find similarities between tasks and effectively learn compressed representations of this distribution.

VI. Conclusion

The over $15\times$ speedups achieved by our jax reimplementions of PCGRL environments allow us to train models for many more timesteps (1 billion) than in previous works (around 200 million). By randomizing map shapes and the placement of pivotal items in initial layouts, we allow for training of more robust and controllable level generator agents. In our experiments, we find that limiting the observation window of trained agents leads to stronger generalization. By evaluating on held-out initial map layouts and constraints, pcgrl-jax can serve as a scalable benchmark for RL agents, with real-world applications for human level designers.

References

[1] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, 2020, pp. 95–101.

[2] S. Earle, M. Edwards, A. Khalifa, P. Bontrager, and J. Togelius, "Learning controllable content generators," in *2021 IEEE Conference on Games (CoG)*. IEEE, 2021, pp. 1–9.

[3] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[4] N. Shaker, J. Togelius, and M. J. Nelson, "Procedural content generation in games," 2016.

[5] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Compu-*

tational Intelligence and AI in Games, vol. 3, no. 3, pp. 172–186, 2011.

[6] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011.

[7] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: Reactive planning and constraint solving for mixed-initiative level design," *IEEE Transactions on computational intelligence and AI in games*, vol. 3, no. 3, pp. 201–215, 2011.

[8] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[9] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications*, vol. 33, no. 1, pp. 19–37, 2021.

[10] M. Guzdial, S. Snodgrass, and A. J. Summerville, *Procedural Content Generation Via Machine Learning: An Overview*. Springer, 2022.

[11] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, "Evolving mario levels in the latent space of a deep convolutional generative adversarial network," in *Proceedings of the genetic and evolutionary computation conference*, 2018, pp. 221–228.

[12] A. J. Summerville and M. Mateas, "Super mario as a string: Platformer level generation via lstms," in *Proceedings of FDG/DiGRA*, 2016.

[13] S. Snodgrass and S. Ontanón, "Learning to generate video game maps using markov models," *IEEE transactions on computational intelligence and AI in games*, vol. 9, no. 4, pp. 410–422, 2016.

[14] I. Karth and A. M. Smith, "Addressing the fundamental tension of pcgml with discriminative learning," in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019, pp. 1–9.

[15] M. Guzdial, N. Liao, and M. Riedl, "Co-creative level design via machine learning," *arXiv preprint arXiv:1809.09420*, 2018.

[16] O. Delarosa, H. Dong, M. Ruan, A. Khalifa, and J. Togelius, "Mixed-initiative level design with rl brush," in *Artificial Intelligence in Music, Sound, Art and Design: 10th International Conference, EvoMUSART 2021, Held as Part of EvoStar 2021, Virtual Event, April 7–9, 2021, Proceedings 10*. Springer, 2021, pp. 412–426.

[17] Z. Jiang, S. Earle, M. Green, and J. Togelius, "Learning controllable 3d level generators," in *Proceedings of the 17th International Conference on the Foundations of Digital Games*, 2022, pp. 1–9.

[18] C. Ye, A. Khalifa, P. Bontrager, and J. Togelius, "Rotation, translation, and cropping for zero-shot generalization," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 57–64.

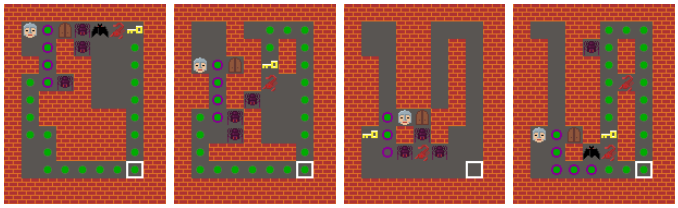


Fig. 7: On the DUNGEON domain the CONV model with full observation mutates the board through a series of playable levels.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[20] R. T. Lange, "gymnax: A jax-based reinforcement learning environment library, 2022b," *URL http://github.com/RobertTLange/gymnax*, 2022.

[21] C. Lu, J. Kuba, A. Letcher, L. Metz, C. Schroeder de Witt, and J. Foerster, "Discovered policy optimisation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 455–16 468, 2022.

[22] M. Matthews, M. Beukman, B. Ellis, M. Samvelyan, M. Jackson, S. Coward, and J. Foerster, "Craftax: A lightning-fast benchmark for open-ended reinforcement learning," *arXiv preprint arXiv:2402.16801*, 2024.

APPENDIX