Learning Controllable Content Generators

Sam Earle
New York University
Brooklyn, New York
sam.earle@nyu.edu

Maria Edwards
New York University
Brooklyn, New York
mariaedwards@nyu.edu

Ahmed Khalifa
New York University
Brooklyn, New York
ahmed@akhalifa.com

Philip Bontrager
TheTake
New York, New York
pbontrager@gmail.com

Julian Togelius
New York University
Brooklyn, New York
julian@togelius.com

***Abstract*—It has recently been shown that reinforcement learning can be used to train generators capable of producing high-quality game levels, with quality defined in terms of some user-specified heuristic. To ensure that these generators' output is sufficiently diverse (that is, not amounting to the reproduction of a single optimal level configuration), the generation process is constrained such that the initial seed results in some variance in the generator's output. However, this results in a loss of control over the generated content for the human user. We propose to train generators capable of producing controllably diverse output, by making them "goal-aware." To this end, we add conditional inputs representing how close a generator is to some heuristic, and also modify the reward mechanism to incorporate that value. Testing on multiple domains, we show that the resulting level generators are capable of exploring the space of possible levels in a targeted, controllable manner, producing levels of comparable quality as their goal-unaware counterparts, that are diverse along designer-specified dimensions.**

***Index Terms*—procedural content generation, reinforcement learning, game AI, pcgrl, conditional generation**

I. Introduction

The idea of using reinforcement learning to learn game content generators—or at least, the successful implementation of this idea—is relatively recent [1]. The basic idea is simple: an agent is trained to construct levels (or other types of game content) in the same way an agent would be trained to play the game.[1] Instead of being rewarded for e.g. winning the game, the agent is rewarded for improving the level.

Compared to using search or optimization methods for content generation [2], Procedural Content Generation via Reinforcement Learning (PCGRL) requires a long training time, but is then able to produce an arbitrary number of content artefacts very rapidly. Compared to methods based on supervised learning [3], [4], PCGRL avoids the need for training data, but instead requires a reward function that reflects content quality.

As PCGRL was only proposed as a methodology very recently, there are many issues that have not been studied yet. One of them is controllability. It is highly desirable for a content generation method to be able to be controlled by the user (such as a human designer or a difficulty adjustment algorithm). For example, one might want to develop a level that favors a specific playstyle, a quest with a certain degree of branching, or a map with a certain balance. The obvious way of controlling the output of an RL-trained content generator would be to change the reward function (just like how you would change the fitness function in search-based PCG), but that would mean retraining the generator for every change of control parameters.

In this paper, we explore approaches to training generators which are controllable *after training*. Or in other words, training single content generator agents to output a variety of content artefacts depending on control parameters. We do this by introducing control parameters as additional "conditional" inputs to the neural network, and then rewarding the generator not only for creating correct artefacts but also adhering to the control parameters. Figure 1 shows an example of generated Sokoban level by one of our trained generators. The generator

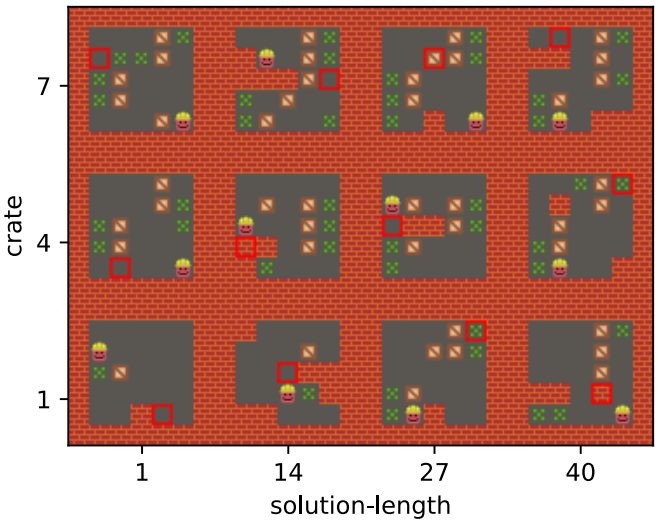

Fig. 1: Generated levels from a trained generator on the game of Sokoban. The generator is controlled during inference to produce small Sokoban levels with variable solution-lengths and numbers of crates.

[1]In this work, "generator" and "agent" are used interchangeably to refer to the level-generating RL agent.

978-1-6654-3886-5/21/$31.00 ©2021 IEEE

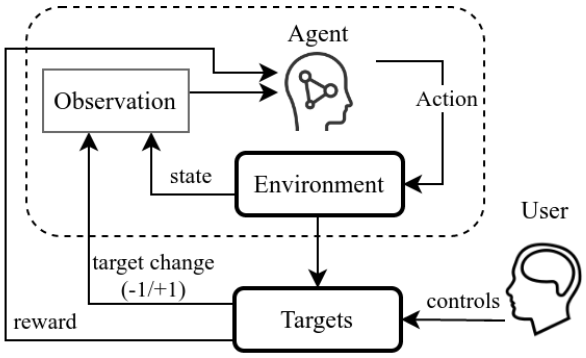

Fig. 2: Controllable PCGRL: In the inner loop, an agent produces content in order to meet certain targets (constraints or heuristics). In the outer loop, a user (or automatic curriculum) modifies these targets.

is able to control the number of crates in the level and the solution-length to provide us with different levels across these two dimensions. It does so with substantial (though not absolute) reliability.

The key to making this work is the control regime. To ensure the agent learns to respond to controls, we need to sample from the space of all possible values these controls may take (e.g. by using a uniform random distribution). But as this space becomes large, this naive approach may become inefficient. For the approach to scale to complex controls, the control regime should focus on targets with the most learning potential for the agent.

Using a curiosity-based teacher algorithm to sample control targets, and the observation and reward scheme described above, we demonstrate controllable PCGRL in three simple 2D level-design domains as well as in free-play modes in SimCity and Rollercoaster Tycoon.[2]

II. Related work

A. PCGRL

The idea behind PCGRL is to frame the PCG process as a game of level design for the RL agent [1]. The notion of building levels in a sequential, goal-driven way complements our intuition about the human level design process, though it is not the approach usually employed by machine learning systems leveraged for PCG [3]–[6]. In PCGRL, the agent is given a random level and can then, turn-by-turn, change tiles in the level as it sees fit. For each change that improves the quality of the level, it is given a reward, and it is penalized when it makes things worse. This approach allows a user with no access to training data to train a functional content generator, by having them instead design a reward function from which an agent can learn. Where ML-based PCG often struggles to produce feasible content, here, functional constraints are communicated to the generator during training via its reward. At the same time, we get the benefits of a neural network-based generator, which can generalize to unseen input to produce novel designs [7]. This allows us to combine strengths of both search-based, and ML-based PCG.

[2]Code is available at https://github.com/smearle/control-pcgrl

In a naive implementation, PCGRL is susceptible to a particular kind of overfitting in which the agent learns to generate the same level every time. We want to avoid this, as we want to learn generators rather than levels. To make the RL agent learn to generate a variety of levels, the original PCGRL paper employed a strategy to make the agent responsive to the initial random level provided at the beginning of each episode. This strategy was designed to limit the number of changes that the agent was allowed to make to the initial map before the level-generation episode terminated (i.e. "game over") during training. The threshold could be set very low, to make the agent very responsive to the provided starting state, or it could be made very high, to give the agent freedom to build a more optimal design. The philosophy behind this strategy was that diversity of agent behavior should be incentivized by the environment, rather than built into the RL model or update rule, so as to remain compatible with any RL training regime. In that same spirit, our approach here is to make PCG Agents controllable by a human designer through changes in the PCGRL framework itself, and not through custom RL algorithms.

Taking advantage of the responsive nature of a PCGRL Agent, there has been followup work to the original PCGRL paper that uses these trained agents as collaborative designers with a human designer [8]. By training the agent with a limited number of allowed changes the agent learns to make very efficient changes which is ideal for working with a human designer where the human and agent take turns. The limitation in this scenario is that the designer cannot communicate intent to the agent and can only communicate through changes made in the environment. In this work we want to make the agent responsive to provided, explicit, design goals, which will hopefully allow for for more meaningful human-AI collaboration.

Contemporaneous with our previous work is the work by [9] and [10]. In [9] the authors devise a game between two game-playing RL agents and a level design agent. This regret minimization game tasks the generator with the production of levels that are neither too hard nor too easy for the players, removing the need for explicitly designing a reward function for level-generation. In [10], the authors also pair playing and generating agents together to remove the need for an explicit reward function, but adapt the level design to the agent mid-game. In each of our experiments, at least one of our controls corresponds to a proxy for player difficulty, and the works cited above demonstrate possible methods for replacing these proxies with actual feedback from learning player-agents in future work. At the same time, the incorporation of arbitrary level-metrics into the reward function gives the human designer additional dimensions along which to control and diversify their levels.

B. Controllable RL

There are several possible approaches to training controllable RL agents. We could train the RL agent to respond behaviorally to text commands [11]. We could train it with different policy networks [12] or memory modules [13] corre-

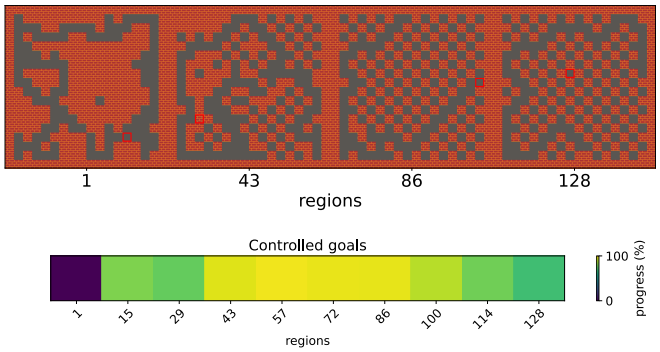

Fig. 3: Generated examples from region controlled agent-generator. A checkerboard pattern is used to efficiently increase the number of disconnected regions, with some corridors remaining to contribute to path-length.

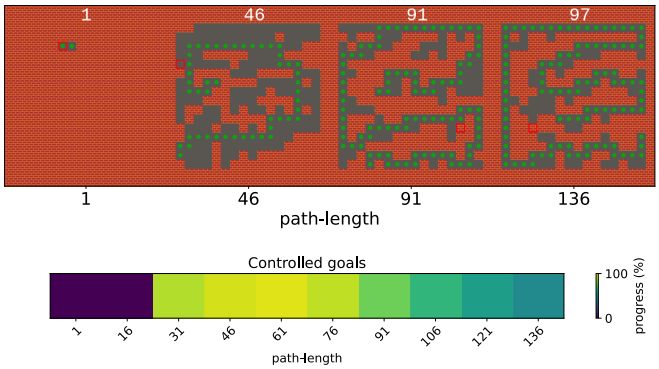

Fig. 4: From left to right: the generator produces paths of increasing length. Its control weakens toward the longest path-lengths, which tend to be learned much later in training.

sponding different goals. Or we could train it with conditional inputs and reward shaping, like in [10], where level designer-agents observe an auxiliary input that corresponds to intended difficulty, and are rewarded for controlling a player agent's level of success on the generated levels.

All of these approaches could potentially be applied to PCGRL (though they have not, yet) and each could address different use cases. In this work we propose an algorithm-agnostic approach involving conditional inputs and reward-shaping. By encoding the goal into the state representation of the level, we allow any RL algorithm to be used and the goal information can be coded with the reward function, without requiring any extra data.

C. Absolute learning progress teacher algorithm

In the BipedalWalker environment, a single RL agent can be trained to navigate diverse terrain, by parameterizing the terrain-generation process with a continuous variable, and sampling this variable from a Gaussian Mixture Model that is fitted to the agent's Absolute Learning Progress during training (ALP-GMM) [14]. The agent's reward is the same across all sub-tasks. The same ALP-GMM teacher algorithm is used in our work to generate a curriculum of content-generation sub-tasks: rather than sampling environments, it samples target behavior (i.e. level characteristics), each of which correspond to a particular conditional input scheme and reward function.

III. Domains

A. Binary, Zelda, and Sokoban

Our main experimental domains are the same ones as in [1] (where they are also described in more detail): Binary, Zelda, and Sokoban. In the Binary domain, tiles can be either wall or floor. The goal is to create the longest shortest path between any two tiles. Zelda is the GVGAI [15] game which is inspired by the dungeon system in the original Legend of Zelda (Nintendo, 1986). Levels need to allow the player character to get the key and open the door, and can place enemies to fight in the level. Sokoban (Thinking Rabbit, 1982), finally, is the classic box-pushing puzzle. A solvable Sokoban level allows for a way in which the player can push each box onto a target.

B. SimCity

In the SimCity reinforcement learning environment based on the open-sourced SimCity 1 [16] (Maxis, 1989), the agent constructs a city by placing zones (residential, commercial, and industrial), infrastructure (road, rail, electricity lines, airports, harbors), and services (police and fire stations, and parks). Citizens will inhabit and develop zones according to local desirability rules and global demand by zone-type, and similarly travel along roads to simulate cycles of commute and recreation.

RL has previously been used to generate player agents that can successfully maximize fixed reward functions comprising weighted combinations of different types of population. The agents here, on the other hand, are trained to build cities resulting in a range of population levels. This is a more plausible approach to generating general playing agents for this type of management sim game, in which artefacts (like cities or theme parks) may be functionally optimal in a multitude of ways.

C. micro-RCT

In the minimal theme-park management learning environment *micro-rct* [3], based on RollerCoaster Tycoon (Microprose, 1999) and reproducing some of the implementation logic of OpenRCT2 [4], the agent builds an amusement park by placing concessions, rides, services (such as washrooms and first aid stalls) and paths. They are frequented by a fixed number of guests (with varying intensity and nausea tolerance thresholds) who produce income for the park. Guests' mood is affected by their experience on rides (according to the ride's characteristics and the guests' preferences and tolerance levels) and internal needs such as hunger, thirst, and bladder.

IV. Method

A controllable RL level generator is trained by feeding it inputs corresponding to target level features, and rewarding it when it produces levels with these features. New values for

[3]https://github.com/smearle/micro-rct
[4]https://github.com/Open/RCT2/OpenRCT2

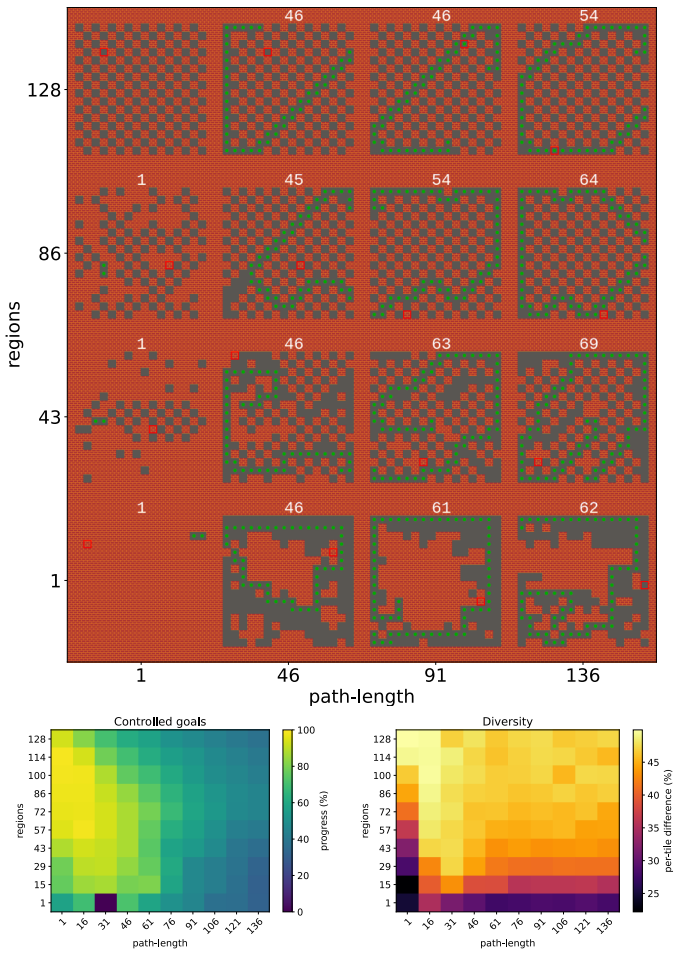

Fig. 5: Generated levels produced by an agent-generator using varying path length and number of regions. It has precise control over the number of regions, but struggles to generate long paths. While it is practically impossible to reach many high path-length and high-regions targets, the agent is also less apt in generating long paths with few regions, likely because paths require more complex behavior to construct than atomic, disjoint regions.

target features are sampled at the beginning of each episode. The episode is terminated when the generator agent either: reaches the target, makes as many per-tile changes as would correspond to 100% of the map, or reaches a limit on the number of steps (equal to the square of the number of tiles on the map).

In this work, we focus on PCGRL's narrow action representation, in which the agent chooses what to build on each tile in sequence, left-to-right and top-to-bottom. This representation performs comparably and has fewer actions than others using additional navigation (turtle) or tile-coordinate (wide) actions.

As in [1], the agent observes a one-hot encoded view of the game board centered around current tile. Additional scalar inputs are concatenated with the agent's 2D observation (channel-wise), corresponding to the direction (-1, 0 or 1) of the target change along some metric. The agent's reward is the amount by which the level has approached (or moved away from) target metrics since the previous step. So, if the agent is generating a maze-like level, where the current length of the maze is 40, and its target length is 20, then in addition to its usual one-hot observation of the map, it also observes a 2D layer filled with -1s representing the desired decrease in path-length, and receives a reward for taking an action that causes such a decrease toward the target.

Let $\mathbf{s_t}$ be a vector representing the user-defined control metrics (e.g. number of enemies, nearest enemy) at time t. At the beginning of the episode, a target/goal vector g is assigned corresponding to desired metrics in the output level. This defines the generator's task, determining both its conditional observation and its reward.

The conditional observation vector, $c = \text{sign}(\mathbf{g} - \mathbf{s_t})$ represents the target directions for each metric. The loss of a given level with respect to the goal vector is given by $l_t = ||\mathbf{g} - \mathbf{s_t}||_{L_1}$. Then the agent's reward at t is $r_t = l_{t-1} - l_t$. The agent is rewarded for edits that close the gap between the level's current metrics and the target, and punished for those that widen it.

Most simply, control-targets can be sampled from a uniform random distribution at the start of each episode. But this may be ill-suited to tasks in which control-metrics are often at odds , where too much training time might be spent attempting to reach impossible combinations of targets.

To make training more efficient, control targets are sampled to maximize the agent's absolute learning progress [14]. If the generator has mastered some sub-space of targets perfectly, the control regime instead favors targets where the generator shows some long term change in performance. This allows the generator to focus on tasks that present the highest opportunity for improvement, as well as those that it may be forgetting.

Generator agents are trained for either up to 500 million frames using PPO [17] as implemented in stable-baselines [18], with content-generation tasks implemented as gym environments [19]. Each experiment is run on a 48-CPU node of a research cluster with no GPU. Typically, Binary, Zelda and Sokoban experiments reach the frame limit after $\approx$ 1, 2, or 5 days, respectively (as Sokoban-generators begin to more frequently produce playable levels, running the solver slows down environment steps considerably). Ranges for control targets are chosen based on estimated lower/upper bounds of these metrics on playable levels (e.g. In Binary, we ask the generator to produce paths from anywhere between 0—no empty tiles on the board—and 136—the length of an optimal zig-zag path on a 16×16 board). Other architectural details and hyperparameters are unchanged from [1]; in particular, initial level-states are sampled randomly using per-tile probabilities that differ by environment.

V. Results

To measure both the agent's progress toward its goals, and the diversity of the levels it generates, the agent is evaluated over a grid in target-space (Figures 3–10). For each cell in the grid, it is allowed one generation episode on each of a shared set of 50 random initial maps. To visualize an array of diverse levels, we select the level resulting in the highest total reward from each cell. Episodes are terminated after 1,000

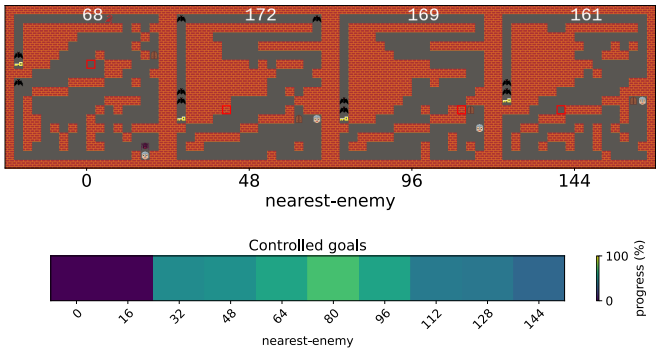

Fig. 6: From left to right: the generator moves the nearest enemy further away from the player. It seeks to maximize the path length of the shortest possible solution (displayed above) in all cases.

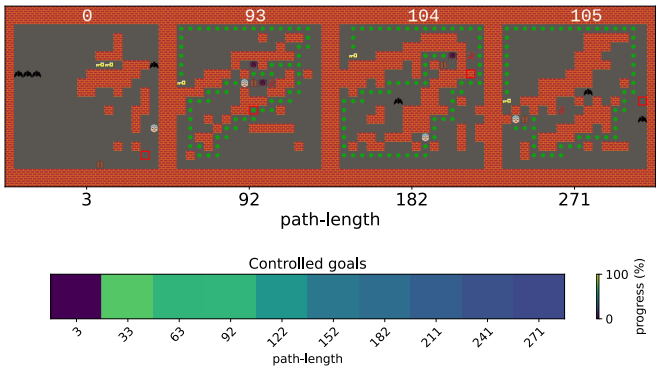

Fig. 7: From left to right: the generator increases the path length of the shortest possible solution, keeping enemies at least 5 tiles from the player.

steps or once the usual termination conditions are met (i.e. the equivalent of 100% of the level is changed or the control targets are reached).

Diversity is computed as the mean per-tile hamming distance between the set of levels generated for a given target. The value is normalized as a percentage, with 0% indicating that all the maps are the same, and 100% that no two maps have the same tile at a given coordinate. Progress is defined as the relative percentage change from the initial state toward the control targets. For example, if the target path length is 20, the initial length is 10, and the agent manages to reach 15 by the end of the episode, then it has made 50% ($\frac{15-10}{20-10}$) progress toward its target. When visualizing results, we restrict mean progress to the interval $[0, 100]$, so that if an agent fails to move toward its target at all, or moves away from it, it is said to have made 0 progress.[5]

A. Original PCGRL environments

1) Binary: In the Binary domain, generators are rewarded for producing levels with long paths and a single region. Here, we train a generator to control for one or both of these metrics. When we control for one metric, the other takes on a fixed value (1 region or maximum path length), and is factored into the reward accordingly (but omitted from the agent's observation).

Generators learn precise control over the number of regions, while attempting to maximize an (equally-weighted) reward for increasing path length (Figure 3). They balance a tightly packed checkerboard pattern with organic corridors of varying length. Maps become increasingly diverse as the number of regions increases and the path-length is forced to decrease, indicating that the generator has learned fewer maps with long paths than it has maps with medium or small paths against a checkerboard background. This is understandable, as there are more constraints on the form which a maximum-length path can take, as opposed to smaller paths which can, for example, be translated about the map to produce many distinct levels.

Conversely, generators can control for specific path length while maintaining a single connected region, producing shorter paths in various forms and locations on the map, and a variety of organic labyrinths to achieve longer paths (Figure 4). During level-generation, path-length tends to grow or shrink gradually. This may allow the generator to more accurately sense whether it has reached or exceeded its target, when its conditional input changes to 0 or changes sign, respectively.

A generator trained to control both regions and path-lengths learns to blend these checker-boarding and labyrinth-growing strategies (Figure 5). It has the most success reaching its goals—and produces the most diverse levels—when aiming for low path-length and varying number of regions.

The generator learns to control regions more easily than path-length presumably because the policy it learns to control regions is simple and highly localized. Still, progress toward longer path lengths is apparent, particularly when the target number of regions is lower (given that increasing regions and path length are conflicting goals). It is also possible that the weights assigned to these respective control metrics are such that it is often optimal for the generator, in terms of its reward, to sacrifice path-length to allow for more regions.

2) Zelda: In Zelda, agents are usually rewarded for placing 1 key, door, and player, between 2 and 5 enemies, maximizing path length (from the player to the key to the door), and leaving a minimum distance of at least 5 tiles between the player and the nearest enemy. To train a controllable generator, we take nearest-enemy and path-length as our control metrics (in particular because they could be taken as proxies for level difficulty), and fix all other targets.

When controlling for nearest-enemy alone, the generator is able to maintain high path-lengths while placing enemies at various minimum distances from the player (Figure 6). These levels tend to share a common structure resulting in high path-length—namely a corridor through which the player must travel back and forth to retrieve the key and access the door—with enemies sliding up and down the corridor between levels to end up closer or further from the player. These levels are consequently more diverse when the closest enemy is nearer the player, since there are fewer constraints on the positioning of any other enemies in the level.

[5]This prevents exploding negative scores when the initial state is very close to the target metrics

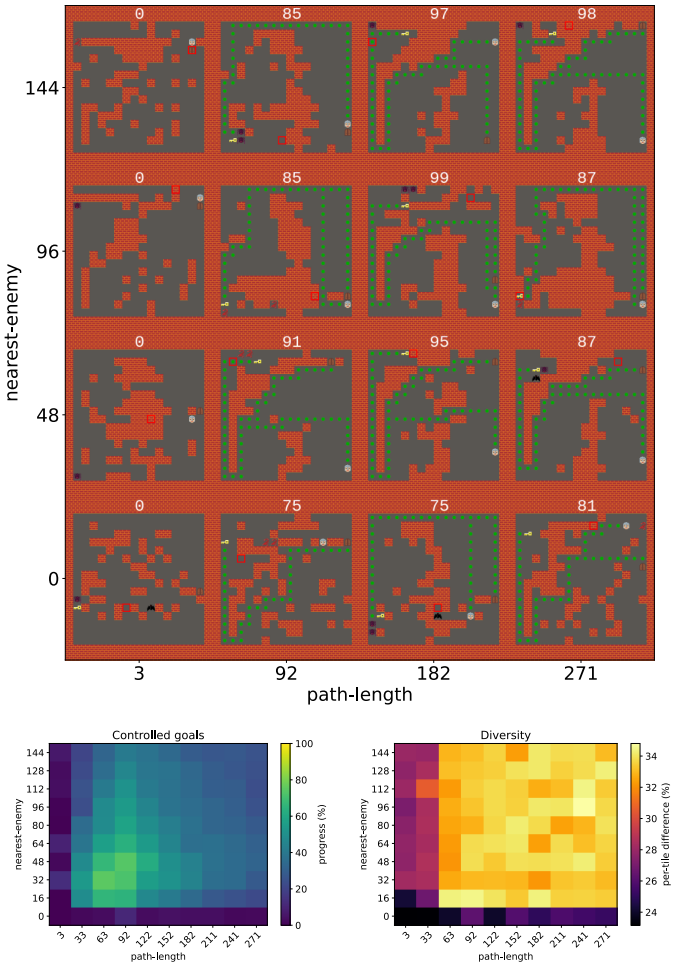

Fig. 8: Left to right: generator increases path length. Bottom to top: generator increases distance to nearest enemy. The generator has the best control over a set of levels in which path-length is around twice nearest-enemy: enemies are placed by the key, with the door close to the player. Other, fixed playability constraints are generally met reliably across the control-space.

Controllable path-lengths are produced by levels ranging from open spaces to more lengthy corridors, with the key and door placed at various distances from the player and each other (Figure 7). Surprisingly, there is greater diversity among higher path-length levels, despite their being more theoretically constrained and thus fewer in number. It may be that while the some trivial family of layouts can be used to easily achieve minimal paths, the generator's learning a diversity of longer-path configurations is crucial to its transitioning gradually and resourcefully from random initial states toward optimal levels.

The generator can also be trained to control both metrics simultaneously (Figure 8). In this case, it tends to favor levels in which the enemy is very close to either the player or the key/door. The agent generates more diverse content when both nearest-enemy and path-length are high or non-trivial.

3) Sokoban: In Sokoban, the default heuristics call for 1 player, at least 2 crates, a matching number of targets, and maximal solution-length (the number of steps taken by an A* solver to push all crates onto targets). Here, the generator

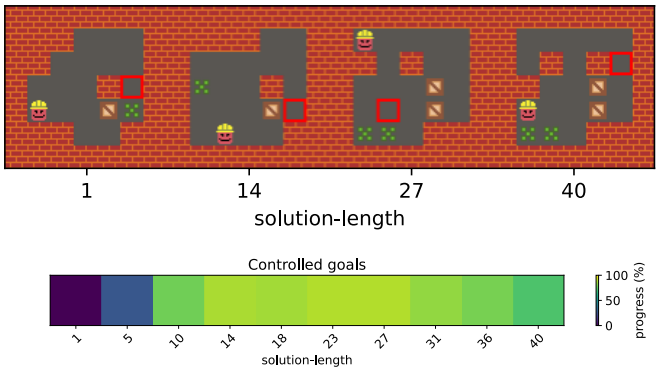

Fig. 9: From left to right: the generator produces small Sokoban levels with increasingly lengthy solutions. Longer solution-lengths require the player to navigate obstacles and backtrack in order to push the crate to the target. Diversity decreases with level complexity.

attempts to control the number of crates and solution-length.

The Sokoban level-generator exhibits strong control over a range of solution-lengths, corresponding to modest complexity on a very small map (Figure 9). The generator reliably produces playable levels, and deviates from its control targets only when these contradict fixed playability targets. It produces a variety of levels, from simple open rooms, to those with simple obstacles, and corridors that require back-tracking before the crate can be pushed toward its target. Most levels use 1 or 2 crates in concert with player backtracking through narrow corridors to control solution-length.

When asked to control the number of crates in addition to the solution-length (Figure 10), the generator instead favors the use of multiple crates to increase level complexity

B. micro-RCT (Roller Coaster Tycoon)

In micro-rct, agents learn to build parks with specified levels of guest (un)happiness (Figure 11). To maximize happiness, a cluster of concession stands is placed by the entrance, and guests immediately buy food/drinks whenever their hunger/thirst falls below a certain threshold. They then receive a happiness boost from the perceived value of the purchased item. To make guests unhappy, the generator places a few small thrill rides by the entrance. The rides are popular among guests, but have a high enough nausea score to induce some vomiting, which causes guests to become disgusted by their cramped surroundings.

C. SimCity

In SimCity, agents learn to control the amount of residential population in the city (Figure 12). A maximal residential population can be achieved without any commercial or industrial zones via a strategic, dense configuration of residential around a single power plant, with disjoint road tiles placed adjacent to zones to increase population density. To induce medium residential population, the agent places zones more disparately, inviting a greater proportion of low-density development. This may allow it to meet lower population targets with increased precision.

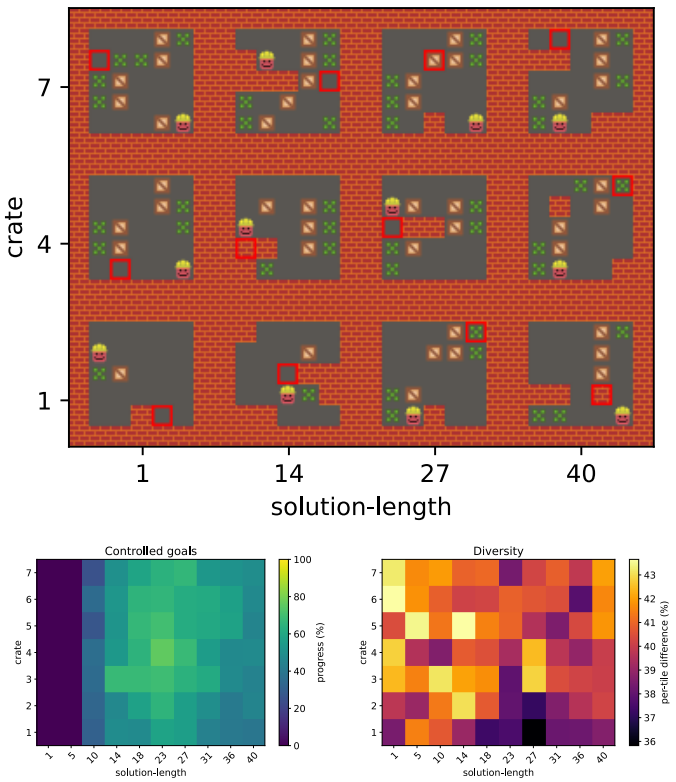

Fig. 10: The generator produces small Sokoban levels with various solution-lengths and numbers of crates. It has the best control over levels with many crates and long solutions. Levels with few crates are the most diverse, as on such a small map, more crates drastically reduces the number of solvable configurations.

D. Control regimes

In Table I, several baseline agents are compared against controllable agents in the Sokoban level-generation domain. A baseline agent allowed to change 100% of the board during training is the most successful during evaluation on a baseline task (in which target metrics are fixed at their default values), consistently outputting near-optimal configurations.

Controllable generators are trained with either a regime of uniform-randomly sampled control targets, or of targets sampled according to learning progress (i.e. an ALP-GMM curriculum). More often than not, agents trained with ALP-GMM show a slight advantage over those trained on randomly-sampled targets (both on the baseline task and on a set of control-tasks). Note that this was not the case, however, in the simpler Binary and Zelda domains, where random sampling of targets tended to outperform or match the performance of an ALP-GMM curriculum, respectively. This discrepancy would seem to support the intuition that the use of adaptive control regimes will be important for scaling up RL-based controllable content generators to more complex domains and control-spaces.

VI. Discussion

Results on level generation tasks show that with conditional (target) input and shaped reward, RL can be used to train generators that encode a set of playable levels that are controllably diverse along measures of interest, such as deterministic proxies for player difficulty. Experiments with management sims suggest that this learned control can handle the stochasticity and temporal dynamics resulting from agent-based simulation. These controls could serve as convenient and useful interfaces for game designers for producing content with given characteristics. They could also act as easily searchable or traversable spaces for the sake of curriculum generation, producing levels adapted to a particular player [20], or for the purpose of training a player agent.

Notably, generators learn to adapt to changing, user-defined targets over the course of long inference episodes, despite having only been trained to approach one set of targets per episode. Novel states can emerge from this traversal of conditional space, and these dynamics set controllable content generators apart from methods that might search for a diverse set of levels directly, arguably giving them a greater degree of expressivity. On the other hand, the computation required to learn a generator is likely to be greater, since this diverse space of levels must be represented in a compressed form in the agent's weights, rather than stored in an archive.

Random initial states during level generation, combined with a limit on the number of changes that may be made to the map, ensure that a controllable agent learns diverse levels for targets where possible. Often, the generator learns to traverse the space of levels in small steps, moving from initial states, incrementally along dimensions of interest until the target is met, thanks to its limited, binary conditional observation and its incentives for efficiency. To ensure paths through this space are explored more thoroughly, and free them of the constraint of the initial random maps, an archive could be kept as in [21].

In cases where there is some trade-off between control targets, the generator is tasked with finding optimal solutions according to some weighting of these controls' effect on reward, which may be difficult to tune. For example, in the Binary problem, it is impossible to produce a maximal number of both paths and regions (because having many regions precludes having a long path). But it may be desirable to search for levels along the pareto front of these multiple objectives. This could be approximated by turning these per-control weights into controls themselves, supplying them directly as conditional input, prompting the agent to learn multiple objectives.

VII. Conclusion

Formulating content generation as a reinforcement learning problem allows us to learn generators that produce game levels of high quality along user-specified dimensions (e.g. path length). But it may also be desirable to have a generator that encodes the space these dimensions produce (e.g. levels of variable path length) rather than one point within it.

We show that this can be achieved by sampling from target points within this space over the course of training. The generator then observes its target and is rewarded for approaching it. The resulting learned generator is controllable,

TABLE I: Sokoban level-generation. Performance of baseline (single-objective) agents, and controllable agents with learning-progress-informed (ALP-GMM) and uniform-random control regimes, with various change percentage allowances. Agents are tested on a baseline task with metric targets fixed at their default values, and on control tasks, in which controllable metric targets are sampled over a grid. At certain change-percentages, controllable agents are competitive with or outperform baseline agents on the baseline task. Controllable agents trained with an ALP-GMM curriculum tend to outperform those trained using a random control curriculum.

			fixed targets		controlled targets			
		evaluated controls	—		solution-length		crate, solution-length	
learned controls	control regime	change percentage	pct. targets reached	diversity	pct. targets reached	diversity	pct. targets reached	diversity
—	—	**0.2**	72	56	41	**57**	36	**55**
		0.6	75	44	47	42	44	44
		1.0	**94**	30	49	29	45	34
sol-length	uniform random	**0.2**	66	51	35	49	29	52
		0.6	76	49	55	44	52	42
		1.0	82	38	60	32	56	32
	ALP-GMM	**0.2**	52	49	25	49	22	45
		0.6	78	51	56	40	50	41
		1.0	83	40	**70**	31	**60**	28
crate, sol-length	uniform random	**0.2**	60	**57**	33	48	32	42
		0.6	81	47	48	45	44	44
		1.0	66	41	49	42	38	45
	ALP-GMM	**0.2**	71	50	37	43	35	44
		0.6	81	38	48	39	43	41
		1.0	73	48	43	40	46	40

Fig. 11: From left to right: the generator produces minimal roller coaster theme parks of increasing guest happiness. Popular but nauseating thrill rides cause some vomiting and decrease happiness, while dense placement of burger stalls increases it. In medium happiness parks, the generator attempt to mitigate nausea-induced unhappiness by placing first-aid stalls throughout the park.

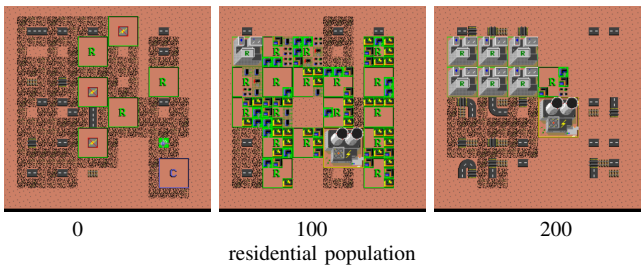

Fig. 12: The generator produces SimCity layouts with increasing residential population. It connects zones to power by adjacency, and uses the presence of adjacent roads to toggle between low and high density to meet its target precisely.

and the user is able to dynamically prompt it to produce levels of a certain type during inference.

To ensure that the reinforcement learning generator trains efficiently, for example by spending less time on large areas of level-space that are impossible to learn (e.g. high regions and path-length), we sample these targets to maximize the agent's absolute learning progress. The generator focuses on those areas where it is excelling or regressing the most, prompting it to explore new and disparate regions of level space while simultaneously recalling what it has already learned.

When procedural content generation is used to learn better game AI, it may be desirable for the level generator to have an interpretable or easily searchable behavior space, bolstering our ability to manually or automatically create curricula of levels for game-playing AI. The approach to learning controllable generators presented here is one such candidate, and our experiments suggest that it can learn to generate a diverse set of playable and complex levels.

The varying aptitude with which these controllable generators are able to explore the space of levels along user-specified dimensions can also help game designers to explore the constraints of level design in their game. And the generator's ability to adapt to changing goals during inference, as well as the relative diversity of levels it can produce within a given set of target features, make it a potentially interesting co-creative tool.

References

[1] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 95–101, 2020.

[2] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[3] 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). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[4] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N Yannakakis, and Julian Togelius. Deep learning for procedural content generation. *Neural Computing and Applications*, pages 1–19, 2020.

[5] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228, 2018.

[6] Philip Bontrager and Julian Togelius. Fully differentiable procedural content generation through generative playing networks. *arXiv preprint arXiv:2002.05259*, 2020.

[7] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*, 2018.

[8] Omar Delarosa, Hang Dong, Mindy Ruan, Ahmed Khalifa, and Julian Togelius. Mixed-initiative level design with rl brush. *arXiv preprint arXiv:2008.02778*, 2020.

[9] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *arXiv preprint arXiv:2012.02096*, 2020.

[10] Linus Gisslén, Andy Eakins, Camilo Gordillo, Joakim Bergdahl, and Konrad Tollmar. Adversarial reinforcement learning for procedural content generation. *arXiv preprint arXiv:2103.04847*, 2021.

[11] Felix Hill, Sona Mokra, Nathaniel Wong, and Tim Harley. Human instruction-following with deep reinforcement learning via transfer-learning from text. *arXiv preprint arXiv:2005.09382*, 2020.

[12] Hossam Mossalam, Yannis M Assael, Diederik M Roijers, and Shimon Whiteson. Multi-objective deep reinforcement learning. *arXiv preprint arXiv:1610.02707*, 2016.

[13] Piotr Mirowski, Matthew Koichi Grimes, Mateusz Malinowski, Karl Moritz Hermann, Keith Anderson, Denis Teplyashin, Karen Simonyan, Koray Kavukcuoglu, Andrew Zisserman, and Raia Hadsell. Learning to navigate in cities without a map. *arXiv preprint arXiv:1804.00168*, 2018.

[14] Rémy Portelas, Cédric Colas, Katja Hofmann, and Pierre-Yves Oudeyer. Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments. In *Conference on Robot Learning*, pages 835–853. PMLR, 2020.

[15] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214, 2019.

[16] Sam Earle. Using fractal neural networks to play simcity 1 and conway's game of life at variable scales. *arXiv preprint arXiv:2002.03896*, 2020.

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[19] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[20] Georgios N Yannakakis and Julian Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, 2011.

[21] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 5(2):e23, 2020.