Procedural Level Generation for Sokoban via Deep Learning: An Experimental Study

Yahia Zakaria, Magda Fayek, and Mayada Hadhoud

Abstract—Deep learning for procedural level generation has been explored in many recent works, however, experimental comparisons with previous works are rare and usually limited to the work they extend upon. The goal of this article is to conduct an experimental study on four recent deep learning procedural level generators for Sokoban to explore their strengths and weaknesses. The methods will be bootstrapping conditional generative models, controllable and uncontrollable procedural content generation via reinforcement learning (PCGRL), and generative playing networks. We will propose some modifications to either adapt the methods to the task or improve their performance. For the bootstrapping method, we propose using diversity sampling to improve the solution diversity, training with auxiliary targets to enhance the models’ quality and sampling conditions from Gaussian mixture models (GMMs) to improve the sample quality. The results show that the generated solutions are more diverse by at least 16% when diversity sampling is used during training. It also shows that training with auxiliary targets and sampling conditions from GMMs can be used to increase the playability percentage. In our experiments, PCGRL shows superior quality and diversity, while the bootstrapped long-short term memory generators exhibit the least control confusion.

Index Terms—Deep learning, generative models, level generation, procedural content generation (PCG), reinforcement learning (RL).

I. INTRODUCTION

Procedural content generation (PCG) is the process of automatically generating content using algorithms. For decades, PCG was a common tool in video games, but it is time consuming and requires a significant effort to build a generator that yields useful, interesting, and diverse content. Even if the generated content is unique, it can still suffer from the “10 000 bowls of oatmeal” problem [1] where every oatmeal bowl is unique but presents the same experience as every other bowl so there is no “Perceptual Uniqueness.”

One approach to PCG is procedural content generation via machine learning (PCGML) [2] where models learn to generate desired content. If the model is a deep neural network, then the algorithm is classified as deep learning. Deep generative models have been developed for a variety of data types (including images [3], text [4], and audio [5]) where they proved to generate convincing results that can be shipped with commercial products. Thus, it is no surprise that deep generative models have been a topic of interest in the field of procedural level generation.

Given the subfield’s novelty, it is rare to find experimental comparisons between methods for multiple reasons. One reason is that articles sometimes target novel problems that it is infeasible to compare their work with previous works. In addition, there are only a few agreed upon methodologies for generator assessment [6]. Moreover, some articles use different benchmarks (different games, dataset, level size, etc.) and metrics to evaluate their methods, thus, their results are hard to compare with other articles.

Our main goal is to present an experimental comparison between a set of recent deep-learning-based generators. As a common benchmark, we pick Sokoban level generation, which was picked as it is unlikely to generate playable levels by chance. The method proposed in [7] randomly generates Sokoban levels while using domain knowledge to eliminate the locations that are unfit to hold goals and crates, yet almost half of the generated levels are unsolvable and less than 10% were deemed interesting. A random generator with almost no knowledge of Sokoban would have a much lower chance of generating a solvable level as we will show in the results.

In addition to employing deep learning and being applicable to Sokoban, only methods that require zero or a few Sokoban levels to initiate the training process are included in our experiments. Based on the aforementioned criteria, our experiments will cover bootstrapping generative models [8], controllable and uncontrollable procedural content generation via reinforcement learning (PCGRL) [9], [10], and generative playing networks (GPNs) [11]. The bootstrapping method in [8] will be used with variational autoencoders (VAE) [12], generative adversarial networks (GAN) [3], variational autoencoder generative adversarial networks (VAE-GANs) [13] (which combines VAEs and GANs), and long-short-term-memory (LSTM) sequence generators [4]. A recent method we skipped was adversarial reinforcement learning for PCG [14] as we have not been able to adapt it for Sokoban level generation yet.

While bootstrapping GANs, we noticed that many of the generated levels could be solved with the same set of actions. Therefore, we propose diversity sampling to enhance the solution diversity of the bootstrapped generators. In addition, we propose training GANs and VAE-GANs with auxiliary targets to increase the playability rate of their output. Finally, we propose
sampling the generator conditions from a Gaussian mixture model (GMM) to increase the chance of generating a playable level. Thus, the contributions of this article are as follows.

1) Adapt the bootstrapping method proposed by [8] and GPNs [11] to Sokoban level generation.
2) Propose diversity sampling as an extension to the bootstrapping method proposed by [8] to improve the solution diversity in the generators’ range.
3) Propose using auxiliary targets to enhance the GANs’ and VAEGANs’ performance.
4) Propose sampling conditions from a GMM to improve the playability of the generated levels.
5) Compare the four methods based on the experimental results.

The rest of this article is organized as follows. In Section II, we mention related works, then we define the level generation problem, state the objectives, and explain Sokoban’s rules in Section III. Section IV briefly explains the bootstrapping method proposed by [8] and propose GMM condition sampling, diversity sampling, and auxiliary targets. Section V briefly explains PCGRL, while Section VI briefly explains GPNs and adapts it for puzzle generation. We detail the experimental setup in Section VII, then show and discuss the results in Section VIII. Finally, Section IX concludes this article.

II. RELATED WORKS

There are multiple previous works that present a comparison between level generators. The submissions to the level generation track in the 2010 Mario AI championship were presented and compared in [15] where the generated levels were judged by human players. In addition, the article included a statistical analysis for the generated levels based on eight key features (number of coins, powerups, gap widths, enemy placement, etc.). Another work [16] also relied on a user study to compare their method with two previous works. More recently, the submissions to the first year of the AI Settlement Generation Challenge in Minecraft have been presented and compared based on the feedback from a panel of human judges [17]. Since most PCG tools aim to provide human players with interesting content, assessment via human feedback is valuable. However, conducting a user study is time consuming, therefore, we rely solely on automated metrics and visualizations such as the expressive range (ER) visualization [18], which can be used to inspect the range of a generator along some chosen metrics of the generated content.

A variety of generative models has been applied to PCG such as LSTMs for Mario level generation [19], GANs for Zelda level generation [8], and VAEs for generating and blending levels from multiple games [20]. Given the scarcity of level datasets, a bootstrapping method proposed by [8] scavenges the model output for playable levels to augment the dataset for training in the upcoming iterations. To distinguish this mechanism from other bootstrapping methods, we will call it “Bootstrapping by Iterative Data Augmentation” or “BIDA” for short. In addition to BIDA, conditional embeddings [8] were proposed to improve the generator’s quality. To train without levels, GPNs [11] learn to generate levels using feedback from an agent trained to play the generated levels. Another approach is PCGRL [9], where an agent learns to modify the level tiles via a reward proportional to the quality improvement. The work was extended [10] to include controls that specify the desired level properties. PCGRL has been used to generate Sokoban levels and the controllable version has been used to control the number of crates and the solution length.

Sokoban level generation is not limited to PCGML and has been tackled using different approaches. A constructive-based approach has been proposed [7] where levels are generated via a random process guided by some rules to increase the chance that they are solvable. Another approach is to generate goal states, then apply reverse playing to search for a state that maximizes a specific heuristic function [21]. Many other constructive and search-based Sokoban generators have been proposed in previous works, but they are not included in this study as they are outside of this article’s scope.

III. PROBLEM DEFINITION

This section presents the problem formulation followed by the objectives we seek to optimize. Finally, we describe Sokoban and the requirements for valid Sokoban levels.

A. Level Generation

A procedural level generator can be abstracted as a function that takes a random sample from the domain and returns a level as shown in (1), where $l$ is the generated level, $G$ is the generator, and $z$ is a random sample from the generator’s domain (e.g., a latent space). In case of conditional generators, a condition $u$, that specifies the desired level properties, is supplied to the generator as shown in (2).

$$l = G(z)$$

$$l = G(z | u).$$

B. Objectives

A good generator should exhibit high quality, high diversity, and high controllability (if it is controllable). In this section, we will discuss the objectives we seek to optimize.

1) Quality: The quality of a generator is determined by the desirability of its generated content. In our comparison, a level is considered desirable if it meets the functional requirements. Instead of adding more constraints to the level desirability, a control-error analysis is added to measure the ability of the generator to output levels with the desired properties. Measuring the generators’ quality is relatively simple since the functional requirements for a level is verifiable via a solver. We use the percentage of playable levels in a generated sample as the measure of a generator’s quality.

2) Diversity: In [8], the lack of diversity was measured as the percentage of duplicate levels. In [10], diversity was measured as the average tile-wise hamming distance between all pairs of playable levels. We argue that both of these have a flaw since usually, we can shift, rotate, flip the level, or change most of its
empty and wall tiles without affecting the level’s solution. Thus, we can have exponentially many levels with high tile diversity that are almost perceptually equivalent. On the other hand, modifying just a single tile can drastically change the solution. We argue that a better measure would be the number of unique generated solutions where a solution is defined as a sequence of actions to reach a goal state. Still, different solutions could boil down to the same high-level plan so they could feel similar from a human perspective. In Section IV-C, We will suggest another option inspired by hierarchical planning. However, we will also report the duplication percentage and the tile diversity since they present important insights into the aesthetic diversity of the generated levels.

3) Controllability: A controllable generator should design levels whose properties are as close as possible to the supplied targets. Therefore, we will report the mean absolute error between the target and actual properties of the generated sample. However, some properties can only be measured for solvable levels (such as the solution length) so the mean absolute error would not consider an unplayable level as a failure to satisfy the controls. So, we suggest using the score function shown in (3) where $l$ are the generated levels, $u$ and $\tilde{u}(l)$ are the requested and actual level properties, respectively, $t$ is the tolerance, and $N$ is the generated sample size. This function is a relaxed version of the accuracy metric where levels, within a certain range of tolerance from the requested properties, are given a partial score.

$$\text{score}(l, u) = \frac{1}{N} \sum_{i=1}^{N} \left\{ \begin{array}{ll} 1 - \frac{\min(t, |\tilde{u}(l_i) - u_i|)}{t} & \text{if } l_i \text{ is playable} \\ 0 & \text{otherwise} \end{array} \right.$$ (3)

C. Sokoban

Sokoban is a deterministic 2-D grid-based puzzle game. The player can move in four directions and push a single crate into an empty or a goal tile. The player wins if every crate is on a goal tile. Sokoban requires long-term planning since actions that lead to dead ends are common.

A Sokoban level is considered compilable if it satisfies the following requirements.

1) There is only one player (no more, no less).
2) The number of crates is equal to the number of goals.
3) All tiles are in one of the seven states shown in Table I.
4) At least one crate is not on a goal tile since we are not interested in levels that are already solved.

A level is considered playable if it is compilable and solvable by the solver used in our experiments.

IV. BOOTSTRAPPING GENERATIVE MODELS

Generative models learn to capture the probability distribution of the training data, so they can overfit on small datasets and mostly generate copies from the training data. To mitigate this issue, a bootstrapping method (BIDA) proposed by [8] augments the training set with new playable levels generated by the model. The augmentation process is run multiple times during training, so the final model is trained on both the original dataset and the playable levels generated by its previous versions.

Alongside BIDA, conditional embeddings were proposed [8] to enhance the model’s ability to respect the tile frequency requirements of playable Zelda levels. Thus, the conditional embeddings were derived from the frequency of each tile in the level. To adapt conditional embeddings for Sokoban, we use different inputs.

1) Walls: The wall count divided by the level’s area.
2) Crates: The crate count divided by the level’s area.
3) Pushed Crates: The number of crates, pushed by the solver, divided by the square root of the level’s area.
4) Solution Length: The minimum number of actions required to solve the level divided by the level’s area.

The walls and crates conditions are inspired by [8]. However, the generator can add crates that cannot be pushed while keeping the level solvable by adding a goal to the same tile. This allows the generator to satisfy the crates condition without any meaningful changes to the level. Therefore, we added the pushed crates count to the conditions to count the meaningfully placed crates since it ignores any crate that does not need to be pushed. The original crates condition was kept since it allows the bootstrapping step to request levels with a diverse number of crates. Other than the walls and the crates, the solution length was added to vary and control the generated level’s difficulty.

A. Generative Models

We test BIDA with four models: GAN [3], VAE [12], VAE-GAN [13], and LSTM [19]. For LSTM, we use it as a sequence generator like how it was used to generate Mario levels [19]. To convert the 2-D level to a sequence, it is traversed in a snake-like pattern (each row is traversed in the reverse order of the previous row).

For GANs, we use the hinge adversarial loss [22] shown in (4) and (5) following [8]. $L_D$ and $L_G$ are the loss functions of the discriminator $D$ and generator $G$, respectively. The real data $x$ are sampled from the training set $P_x$, the latent vector $z$ is sampled from a standard normal distribution $\mathcal{N}(0, I)$, and $u$ is the condition vector.$$
L_D = \mathbb{E}_{x \sim P_x}[\max(0, 1 - D(x|u))] + \mathbb{E}_{z \sim \mathcal{N}(0, I)}[\max(0, 1 + D(G(z|u)|u))] \tag{4}
$$
$$
L_G = -\mathbb{E}_{z \sim \mathcal{N}(0, I)}[D(G(z|u)|u)]. \tag{5}
$$

For VAEs, the objective is to optimize the evidence lower bound loss shown in (6). To avoid confusion with the discriminator (which is denoted by $D$), we will use $G$ to denote the
decoder. The same loss \( L_E \) and \( L_G \) is applied to the encoder \( E \) and decoder \( G \). The reconstruction term uses the cross-entropy loss \( \mathcal{H} \) over the tiles. The latent vector \( z \) is sampled from the distribution predicted by the encoder.

\[
L_E = L_G = -\mathbb{E}_{x \sim P_r} \left[ \mathcal{H}(x, G(z|u)) + D_{KL}(E(x|u) \parallel \mathcal{N}(0, I)) \right], \quad z \sim E(x|u). \tag{6}
\]

VAEGANs combine GAN and VAE losses but replace the tile-wise reconstruction loss with a feature-wise loss where the feature vector \( h_G \) is extracted from the discriminator’s last hidden layer. Like the original VAEGAN, our motivation is to incentivize reconstructions that are similar feature wise rather than tile wise. The VAEGAN losses are shown in (7)–(9) where the parameter \( \alpha \) controls the weight of the feature error loss in the generator loss.

\[
\begin{align*}
L_D &= \mathbb{E}_{x \sim P_r} [\max(0, 1 - D(x|u))] \\
&+ 0.5 \mathbb{E}_{x \sim P_r} [\max(0, 1 + D(G(z|u)|u))] \\
&+ 0.5 \mathbb{E}_{z \sim \mathcal{N}(0,I)} [\max(0, 1 + D(G(z_n|u)|u))]

z_r &\sim E(x|u) \tag{7}
\end{align*}
\]

\[
\begin{align*}
L_G &= 0.5 \mathbb{E}_{z \sim \mathcal{N}(0,1)} [-D(G(z|u)|u)] \\
&+ 0.5 \mathbb{E}_{x \sim P_r} [-D(G(z_r|u)|u)] \\
&+ \alpha |h_D(x) - h_D(G(z_r|u))|^2, \quad z_r \sim E(x|u) \tag{8}
\end{align*}
\]

\[
\begin{align*}
L_E &= -\mathbb{E}_{x \sim P_r} [\|h_D(x) - h_D(G(z_r|u))\|^2] \\
&+ D_{KL}(E(x|u) \parallel \mathcal{N}(0, I)), \quad z_r \sim E(x|u). \tag{9}
\end{align*}
\]

\section*{B. Sampling Conditions From a GMM}

To sample from a conditional model, we need a latent vector and a condition. While latent vectors are sampled from a standard normal distribution, conditions have no parameterized distribution to sample from. The first option is to sample conditions from a uniform distribution. However, some regions in the distribution could be impossible to satisfy (e.g., one crate only with a solution of 100 steps). The second option is to sample from the dataset, but this requires bundling all the training conditions (thousands of vectors) alongside the deployed model. Also, it provides no convenient way to sample conditions given a fixed set of user-supplied targets (e.g., the user may want to specify the difficulty but not care about the number of crates, walls, or player position). Therefore, we propose fitting the level properties of the final training dataset into a GMM. It is much more compact than storing all the conditions and can be sampled unconditionally and conditionally. It is noteworthy that we do not use GMMs for bootstrapping.

\section*{C. Diversity Sampling}

In BIDA, the model is trained on the dataset augmented by its previous versions’ playable output. So, if at any time, the model collapses into a single mode (which is likely), the upcoming training distributions will drift toward that mode causing a snowball effect. Our first attempt to mitigate this issue was by sampling conditions for augmentation from a uniform distribution \( U(\min u, \max u), u \in \text{training data conditions} \) rather than from the dataset to avoid oversampling from modes. However, the generator still created levels with almost similar solutions.

Therefore, we propose diversity-sampling based on the level solution where the training data are grouped into clusters of similar solutions. To create a training batch for the model, we uniformly sample clusters, then randomly pick levels from them. The motivation is to show the model a diverse view of the solutions available in the dataset. To decide if two solutions are similar or not, we use a distilled version of the solution, that we call a “Signature,” where solutions are considered similar if they have the same signature. To compute the signature, we group together primitive actions into high-level actions (HLA), where each HLA translates to “go to a crate and push it a number of times in a certain direction.” Then, we only keep the push directions and ignore the other parameters (the crate and the number of pushes) to simplify clustering. To make the signature rotation and flipping invariant, we rotate it till the first action is “R.” After that, if there are any vertical actions and the first one is “D,” we flip the signature along the y-axis. This representation has some issues; for example, it does not take into consideration that crate is being pushed by each HLA so levels with different crate counts could have the same signature. While there are probably better representations, signatures were enough to significantly increase the unique solutions in the generated samples. Searching for better options is left for future work.

\section*{D. Auxiliary Targets}

GANs learn to mimic the training data distribution. However, as level generators, they should not only capture the dataset’s aesthetic but also the functional requirements (e.g., the number of goals and crates must be equal). When diversity sampling was applied, we noticed a decrease in the percentage of compilable levels in the generated output. A possible explanation is that, rather than capturing the functional requirements, GANs lean toward learning tile patterns that should be harder with a more diverse training dataset. So, we propose adding an extra target to the discriminator (inspired by Auxiliary Classifier GAN [23]) where it has to differentiate between compilable and uncompilable levels in real and fake data. In addition, we add regression targets for the object count (walls, players, crates, and goals) in the level since they are crucial for detecting the level compilability.

\section*{V. Procedural Content Generation via Reinforcement Learning (PCGRL)}

PCGRL [9] formulates level generation as a Markov decision process where the agent’s actions modify the level, and the reward is proportional to the level’s quality improvement. A PCGRL environment has multiple representations out of which we will focus on the following.

1) **Narrow:** The agent can modify the tile at its location while moving across the level in a scanline fashion.
2) **Turtle:** The agent can modify the tile at its location, but it can instead move to an adjacent tile.
3) \textit{Wide}: The agent can modify any tile at any location in the level. The agent has no location.

Controllable PCGRL [10] extends the observation by adding extra channels containing $+1$, $-1$, or $0$ to specify whether the corresponding level property should increase, decrease, or stay still, respectively. For \textit{Sokoban}, the controllable properties are the number of crates and the solution length.

The PCGRL reward function for \textit{Sokoban} motivates the agent to create a solvable level with one connected region, a crate count within a specific range, and to increase the solution length. In controllable PCGRL, the reward function motivates creating levels with the specified targets.

To maintain diversity, PCGRL limits the number of allowed changes. Otherwise, the agent would overwrite the whole input and memorize a single high-reward level. On the other hand, controllable PCGRL left the limit at 100\% of the map.

For training PCGRL on \textit{Sokoban}, the bottleneck is the reward calculation since it requires the solution length. \textit{Sokoban} solving is time consuming since the search space is usually huge. The first optimization was to cache the solutions since Turtle and Narrow agents spend most of the time moving rather than applying changes. Caching almost halved the training time but it was still slow. Therefore, we had to rewrite the solver using the C programming language which ran $400\times$ faster than the original Python solver.

VI. GENERATIVE PLAYING NETWORKS (GPNs)

GPNs [11] are like GANs but the discriminator is replaced by a reinforcement learning (RL) agent and the generator trains to minimize the absolute of the generated level’s predicted value. In other words, it trains to generate hard yet playable levels, while the RL agent trains to win the generated levels in the least number of steps. Although GPNs do not require a dataset, the agent can be pretrained on a few levels to facilitate the learning.

In GPNs, the agent cannot be trained using the original reward received from the game environment, so the authors in [11] propose a simple reward function where the agent receives 1 if it wins, $-1$ if it loses, and 0 otherwise. It also proposes a set of possible modifications to the reward function that can be applied based on the game. We found that the best results for the \textit{Sokoban} level generation were obtained by applying the following two modifications: giving the agent a small reward whenever the environment returns a positive reward (which happens when a crate gets pushed onto a goal) and by decreasing the winning reward proportional to the episode length. The second modification presents two benefits: the agent has more incentive to solve the level faster and the generator has more incentive to generate levels that require more steps to solve. The reward function used in our experiments is as shown in (10), where $t_n$ is the episode length, $t_{max}$ is the maximum allowed episode length, and $r$ is the reward returned by the \textit{Sokoban} environment ($+1$ whenever a crate is pushed onto a goal tile).

\[
R_{\text{puzzle}}(S_n) = \begin{cases} 
1 - \frac{t_n}{t_{max}} & \text{if player wins} \\
-1 & \text{if player loses} \\
\frac{1}{t_{max}} & r > 0.
\end{cases} \tag{10}
\]

VII. EXPERIMENTAL SETUP

In this section, we introduce the dataset and discuss the network architectures used by the four methods. Finally, we detail the rest of the training and generation configuration.

A. Dataset

The dataset shown in Fig. 1 was used as the initial dataset for BIDA and for pretraining the GPN agent. It contains 12 levels with varying degrees of difficulty. The experiments do not need a large dataset since none of the methods require a large dataset to initiate the training process. Since the training process (for BIDA and PCGRL) will require solving many levels, the level size $7 \times 7$ was picked as a compromise between leaving room for generating interesting levels and keeping the solving time within a reasonable limit. While pretraining the GPN, we randomly flip and rotate the level after loading it. When used for BIDA, the dataset is expanded by adding the intermediate states along the solution path of every level into the dataset to augment it with varying values for the solution length condition. The expanded dataset contains 167 levels with 145 unique solutions and 16 unique signatures. The maximum solution length found in the dataset is 31 steps.

B. Network Architecture

For BIDA, we use the same generator architecture for GAN, VAE, and VAEGAN, which contains four hidden transposed convolutions (output sizes: 64, 64, 32, and 32, and kernel sizes: $5 \times 5, 3 \times 3, 3 \times 3$, and $3 \times 3$) each followed by instance normalization. After every two hidden layers, we add a self-attention layer. The encoders and discriminators are also the same except for the last layer. The latent size is 128. In addition, we use spectral normalization on every convolution layer in the discriminator. A skip connection and a trainable parameter $\gamma$ was added to the self-attention module as shown in Fig. 2 to match the original self-attention GAN [24]. All the internal activation functions are leaky ReLUs with 0.2 negative slope. The four conditions are treated similarly in all modules where it
The tiles are selected randomly with some weights. Levels We used a residual height steps with the Naive: First, a random integer 1 iterations with height iterations.

For PCGRL, we follow the agent architectures from [9] with one more convolution layer (3 × 3) and ReLU activation added to each network. For the GPN, the agent and the generator networks were redesigned to fit each network. For BIDA, the models are trained for 2 × 10^4 iterations with 32 batch size and RMSProp [27] at a learning rate of 10^{-3}. Every 100 iterations, we scavenge new playable levels from a generated sample of 128 levels. For VAEGAN, α is 10^{-2}. For auxiliary targets, the compilability loss is cross entropy, while the regression losses are L1 with weights 1/49, 1/5, 1/5, and 1 for the walls, goals, crates, and players, respectively. For fitting the conditions to a GMM, we used a Bayesian GMM [28] with 16 components. To verify the levels, we use breadth-first search with a limit of 10^6 iterations.

In all the PCGRL experiments, the agent trains for 5 × 10^8 steps where the solver also uses breadth-first search with a limit of 10^6 iterations. The tile set was constrained to only tiles with a single object, otherwise the agent’s performance severely declines as it gets stuck in a local optimum where it usually adds a goal in the same tile with every crate. The agents were trained using proximal policy optimization [29] with Adam [30] at a learning rate of 10^{-4}. We use separate networks for the actor and the critic, since they yielded better results compared to weight sharing. The change percentage was set to 40% and 100% for uncontrollable and controllable PCGRL, respectively. During generation, the change percentage was set to 100% for both. For uncontrollable PCGRL, the maximum crate count was 10 and the termination condition is to achieve a solution length greater than or equal 100. For controllable PCGRL, the targets’ ranges were [1,10] for the crates and [1,100] for the solutions. We test each agent twice: once as a deterministic policy and once as a stochastic policy. For deterministic policies, the episode terminates as soon as a state is revisited as it means the agent is stuck in a loop.

For GPN, we pretrain the agent for 2 × 10^7 steps with the episodes limited to 50 steps. After that, the GPN is trained for 100 epochs where in each epoch, the generator is trained to optimize the utility for ten iterations, then trained to increase the diversity for 90 iterations followed by training the agent for 10^6 iterations with a 50% chance of sampling the training level from the elites. The generator’s batch size is 128 and the latent size is 512.

As a baseline, we report the results of two random generators. Their designs encode as minimal knowledge about the game as possible so that we can confirm that the other generators are performing better than random. These random generators work as follows.

1) Naive: The tiles are selected randomly with some weights. The marginal player probability is 1/(width × height), and the marginal crate probability (and similarly for goals) is 1/max(width, height). The goal probability is independent of having a player or a crate in the tile. The wall probability is half the probability of not being a player, a crate, or a goal.

2) Compilable: First, a random integer C is sampled uniformly from the range [1, max(width, height)]. Then, C locations are randomly selected for the goals and C + 1 locations are sampled where the first location is for the player and the rest are for the crates. From the empty locations, we randomly select locations to be walls where the number of the walls is uniformly sampled from 0 to the number of empty locations.

In addition, we include the output of the generator described in the imagination augmented agents (I2A) article [21] in our comparisons. The I2A generator is capable of generating levels that cover a wide range of properties (solution length and crates) so it is included as an additional point of reference to help put the compared methods into context. However, it is not included as a representative of search-based methods, which is out of this article’s scope. During generation, the number of crates is sampled uniformly from the range [1, 10].

For evaluation, each generator creates 10 000 levels from random inputs. To evaluate the controllability, we request 1000 levels for every crate value in the range [1,10] and 100 levels for every solution-length value in the range [1,100]. While testing each control, the other controls are sampled from a GMM given the requested controls. Since PCGRL has no training data, we generate a sample of 10 000 levels and use the solvable levels to generate a GMM for the PCGRL agent. While sampling controls for PCGRL, we clamp the sampled controls to the range used during training. Since generative models are fast to train, we run each configuration five times then we report the mean and the standard deviation of the results. All the experiments were conducted on a machine equipped with a 16 core CPU and an Nvidia RTX 3090 GPU.
TABLE II
QUALITY AND DIVERSITY STATISTICS FROM A SAMPLE OF 10 000 LEVELS GENERATED BY EACH METHOD

| Method | Quality (Average ± Standard Deviation) | Diversities (Average ± Standard Deviation) | Playable Levels | Unique Solutions | Unique Signatures |
|--------|---------------------------------------|--------------------------------------------|----------------|-----------------|------------------|
| Random | 0.0%                                  | 0.0%                                       | 20.0% ± 44.7%  | 0.0%            | 0.0%             |
| Naive  | 40.5% ± 21.0%                         | 42.6% ± 4.9%                              | 43.0% ± 0.07   | 195.0 ± 53.0    | 127.6 ± 45.6     |
| LSTM   | 42.5% ± 31.1%                         | 24.9% ± 4.4%                              | 49.6% ± 11.4%  | 303.6 ± 11.2    | 209.0 ± 15.0     |

For GAN, VAEGAN, VAE, LSTM, the results are aggregated from the five runs of the training experiment.

VIII. RESULTS AND DISCUSSION

In this section, we show and discuss the results and compare the methods in each of the following aspects: quality, diversity, training time, generation time, controllability and tile distribution, and entropy.

A. Quality and Diversity

Table II shows the statistics related to each generator’s quality and diversity. GMM, DS, and AT denote Gaussian mixture models, diversity sampling, and auxiliary targets, respectively. The symbol ~ means that the generation was guided by a stochastic policy. Copies from dataset are the percentage of playable levels copied from the initial dataset. Tile diversity is the average hamming distance between all pairs of playable levels. The dataset statistics are included as a reference while some results were omitted (other combinations of DS, AT, and GMM) to save space. Among the random generators, the compilable level generator created 458 playable levels with 457 unique solutions and 151 unique signatures that outperforms the solution diversity of many bootstrapped models despite having only a few playable levels. The naive generator rarely generates a playable level, and in the batch generated for the comparison, none of the levels were playable. The I2A generator results show the limitations of our solver since it solved only 89.5% of the generated levels although the I2A generator is guaranteed to generate playable levels.

For GANs, VAEGANs, and LSTMs, the first row in their table section includes the results of running without BIDA or any other modification. For GAN and VAEGAN, training without BIDA decreases both the quality and the diversity. For VAE and LSTM, the playability percentage is higher without BIDA but at a significant decrease in unique signatures. For all the models, the duplicates and copies are significantly higher without BIDA. However, it should be noted that with BIDA only, both GANs and VAEGANs still generate less unique solutions than the compilable random generator. This can be attributed to the fact that all the models trained with BIDA are trained to mimic the distribution of the training data with no incentive to generate levels with new solutions. Thus, the solution diversity is usually limited by the number of unique solutions discovered while training. We tried increasing the training steps to enhance the diversity, but the model performance only increased to a certain level before starting to degrade as the number of unique signatures in the training data increased.

From the results, we can notice the following.
1) Sampling conditions from a GMM increases the playability percentage. This holds true even for the omitted results. Also, it usually decreases the tile diversity.
2) Diversity sampling always increases the unique solutions and signatures. However, it usually increases the duplication rate and decreases the playability percentage.
3) Training with auxiliary targets, when used with diversity sampling, increases the playability percentage. It should
Table III

| Name  | AT  | Training Time | Call Time | Calls / Level | Level Generation Time |
|------|-----|---------------|-----------|---------------|-----------------------|
| GAN  | ✓   | 11:12s ± 00:24s | 1.09ms    | 1             | 1.09ms                |
| VAE  | ✓   | 13:55s ± 00:14s | 1.09ms    | 1             | 1.09ms                |
| LSTM | ✓   | 19:56s ± 00:07s | 1.09ms    | 1             | 1.09ms                |
| VAE  | ✓   | 24:15s ± 00:12s | 1.09ms    | 1             | 1.09ms                |
| VAE  | ✓   | 6:51s ± 00:11s  | 1.09ms    | 1             | 1.09ms                |
| LSTM | ✓   | 16:02s ± 00:32s | 0.23ms    | 49            | 11.27ms               |

Calls per level represents the number of calls required to generate one level.

Fig. 3. Samples of Generated Levels to demonstrate the effect of diversity sampling. (a) GAN with auxiliary targets but without diversity sampling. (b) GAN with auxiliary targets and diversity sampling.

be mentioned that training with auxiliary targets, without diversity sampling, decreased the playability rate.

Among the models trained with BIDA, VAEGANs have the highest playability percentage but is outperformed by GANs, VAEs, and LSTMs in tile diversity, unique solutions, and signatures. Both VAEs and LSTMs outperform GANs in generating unique solutions and signatures. In addition, LSTMs have higher playability than GANs. Regarding tile diversity, LSTMs outperforms the other models.

Fig. 3 shows a random sample (without duplicates) of levels generated by two GANs: one is trained with diversity sampling while the other is not. While tiles vary in both samples, the solutions vary significantly when diversity sampling is applied. It should be noted that Fig. 3 was sampled from the least diverse model of each configuration and that even without diversity sampling, it is not common that four random samples would end up with the same solution.

Fig. 4. Samples of solvable levels generated by the GPN.

For uncontrollable PCGRL, all agents generate high quality levels with significantly more unique solutions and signatures than any model trained with BIDA, and they even exceed the I2A generator. This shows an advantage of PCGRL where the diversity stems from limiting the changes to random levels, rather than mimicking a distribution like the BIDA models. In general, acting stochastically improves the results for all agents. The agent with the highest playability rate changes depending on whether the policy is deterministic or stochastic, but when the policy is stochastic, the three agents have almost the same playability percentage. Since the three agents have almost similar performance, we picked the turtle representation to train the controllable PCGRL agent since it required the least training time as shown in Table III.

In most of the cases, the controllable PCGRL agent achieves a lower playability rate and solution diversity compared to the other PCGRL agents. The decrease in diversity could be attributed to the fact that the controllable PCGRL agent was allowed to change up to 100% of the level tiles. The decrease in the playability rate could be attributed to the added complexity of having to satisfy the supplied controls. Overall, it still generates significantly more unique signatures than any models trained with BIDA.

For GPNs, the playability percentage is not low, however, most of the levels are duplicates and both the tile and solution diversity are very low. By inspecting a generated sample shown in Fig. 4, it seems that the levels are minor modifications of two
levels from the original dataset. By testing the GPN agent, it only solved the five trivial levels (out of the 12 dataset levels) among which two levels required the most steps. Thus, the generator prefers to imitate these two levels. To improve the agent’s performance, we attempted to change the agent to use deep repeated convolutional LSTM (DRC-LSTM) [31]. Although this change improved the agent’s performance, the generator failed to learn. Therefore, more work is needed to tune the GPN architecture for Sokoban and to explore the effect of the agent’s architecture on the generator’s learning process. However, given the current results, we did not attempt to train a GPN from nothing.

Fig. 5 shows the ER of every method, which are generated as described in [18]. The axes represent the solution length and the number of crates pushed by the solver. For the GANs, VAEGANs, VAEs, and LSTMs, we plot the average of the five runs. Compared to the initial dataset distribution in Fig. 5(a), all the generative models have ranges that expand beyond the initial dataset. Compared to the compilable random generator, all the methods cover more range except for the GPN and the VAEGAN models.

Although most methods have good coverage over the solution length, they rarely generate levels with more than seven crates. The I2A generator’s ER shows that the solver can solve levels with more than seven crates, but it is still intuitive that keeping the crate count low increases the chance that the solver can solve the level. This presents an incentive to keep the crates count low, especially for the uncontrollable PCGRL agents where they have no incentive to increase the number of crates since they learned to generate levels with long solutions using only a few crates. The controllable PCGRL agent has an incentive to
generate levels with up to ten crates (which is the range from which the crate control is randomly sampled), but it still fails to generate levels with more than seven crates. It is noteworthy that in an unreported experiment, where the solver contained a greedy best-first search stage, the controllable PCGRL agent was able to generate levels with up to ten crates, but it covered solution lengths up to 70 steps only.

For GANs and LSTMs, diversity sampling shifts the mode from the bottom-left bin (the mode in the expanded dataset) to a higher solution length and crate count. For GANs and VAEGANs, diversity sampling helps expand the ER, but for VAEs and LSTMs, diversity sampling slightly decreases the coverage over higher solution lengths.

B. Training and Generation Time

Table III shows the training and generation time for each generator. Training with BIDA is significantly faster where it finishes in less than 30 min, while PCGRL and GPN require no less than a day. We did not include separate results for diversity sampling since it had no effect on the training time. For PCGRL, the training time mainly depends on the frequency of edit actions, so the narrow and wide agents took more time since their action space contains edit actions only. Although the GPN is trained for significantly fewer steps, it took more than a day due to the larger agent architecture and the relatively slow GVGAI environments.

Table III also includes the times for a single network evaluation call, which is calculated by supplying a random input (batch size = 1) for $10^4$ iterations. GANs, VAEs, and VAEGANs have the same time since they share the same generator architecture. The LSTM network is smaller and faster to run but it requires more calls to generate a level since it generates one tile per call. While acting deterministically, the wide agent requires the least steps, since it does not need to move to change a tile. While acting stochastically, the controllable turtle agent requires the least steps. Overall, deterministic policies finish fast since they are terminated as soon as they get stuck in a loop. Overall, GANs, VAEGANs, VAEs and GPNs generate levels faster since they require only one call per level batch. However, the iterative nature of PCGRL is useful in certain applications such as mixed initiative PCG.

C. Controllability

Table IV shows the control error for controllable models. The error is the mean absolute difference between the requested and the actual level property and the score is computed as shown in (3). The error is computed for playable levels only since it is impossible to compute the pushed crates and solution length for unsolvable levels. For the control score, we use a tolerance of 2 and 10 for the pushed crates and solution length properties, respectively. The PCGRL agent is tested twice: once where the untested control is sampled equally from all its possible values and once where it is randomly sampled from a GMM.

Next, we visualize the confusion matrices for the pushed crates in Fig. 6 and for the solution lengths in Fig. 7. The matrices include all the generated levels, where the first column is dedicated for unplayable levels.

Fig. 6 shows that VAEs and LSTMs have the least crate confusion followed by GANs with diversity sampling. For GANs and VAEGANs, diversity sampling seems to decrease the crate confusion while auxiliary targets increase it. The PCGRL agent has some control over the crates, but the most probable output for every input lies around the range 3–5 crates.

For the solution length, VAEs and LSTMs also exhibit the least confusion as seen in Fig. 7. The effect of diversity sampling is different here. For GANs and VAEGANs, diversity sampling seems to increase the solution-length confusion while for VAEs...
Fig. 6. Crates confusion matrices (DS: Diversity sampling and AT: Auxiliary targets). The y-axis represents the requested pushed crates count, and the x-axis represents the actual number of crates pushed by the solver. The first column is reserved for unsolvable levels.

Fig. 7. Solution length confusion matrices (DS: Diversity sampling and AT: Auxiliary Targets). The y-axis represents the requested solution length, and the x-axis represents the actual solution length. The first column is reserved for unsolvable levels.
and LSTMs, it decreases it. An interesting difference between PCGRL and the other models is their response to requesting a high crate count or solution length; PCGRL tends to successfully return a playable level but usually with a lower value than requested, on the other hand, the other models tend to fail at generating a playable level.

D. Tile Distribution and Entropy Analysis

One interesting observation of the generated levels is that they tend to have random tiles that serve no purpose in the level, which is rare in human designed levels. There are some properties that could differ between procedurally generated levels and human designed levels, which we are interested in exploring. So, the following three statistics are presented in Table V, which are as follows.

1) Entropy: It is used to measure randomness in the levels. It is computed using the frequency of tile patterns \((size = 2 \times 2)\) collected using a sliding window of stride 1. The entropy is calculated for each level separately, then the mean and standard deviation over all the playable levels is presented. A higher value should indicate a more haphazard tile placement.

2) Empty tiles: It is the percentage of empty tiles (contains no objects) in each level aggregated over all the playable levels. Adding more empty tiles should increase the chance that the level is solvable so it is likely that some generators would learn to prefer levels with more empty tiles.

3) Idle crates: It is the number of crates that are not pushed by the solver. It is rare to find idle crates in human generated levels, so we are interested in how much the generators differ in that regard. It should be noted that the PCGRL agent cannot add crates with goals in the same tile so the number of idle crates will always be 0. Although the original dataset contains no idle crates, the expanded dataset has some semisolved levels with some idle crates.

The results in Table V shows that almost all the methods have higher entropy output compared to the expanded dataset. An exception is the LSTM generator (without diversity sampling), which has almost the same entropy as the expanded dataset. On the other side, the compilable random generator has the lowest entropy since its levels contain a lot of empty regions. This can be seen in the percentage of empty tiles where it has the highest percentage among all the generators. It is followed by the PCGRL agents where more than half the tiles are empty. The models trained with BIDA also tend to add more empty tiles compared to the initial dataset with VAEGAN models being the only exception. Regarding the idle crates, they appear in every model trained with BIDA but at a low rate. They most commonly appear in GANs (except when diversity sampling is used without auxiliary targets) and LSTMs (without diversity sampling).

### Table V

| Name     | DS   | AT     | Entropy | Empty Tiles | Idle Crates |
|----------|------|--------|---------|-------------|-------------|
| Random   | Compilable | 3.3 ± 0.8 | 77.8% ± 10.7% | 0.2 ± 0.5  |
| Search   | 12A [21] | 3.8 ± 0.7 | 31.3% ± 11.3% | -           |
| Dataset  | Expanded | 3.9 ± 0.4 | 43.7% ± 10.7% | 0.4 ± 0.5  |
| GAN      | ✓    |        | 4.2 ± 0.3 | 50.4% ± 16.5% | 0.9 ± 0.2   |
| VAEGAN   | ✓    |        | 4.3 ± 0.3 | 49.7% ± 13.9% | 0.2 ± 0.5   |
| VAE      | ✓    |        | 4.3 ± 0.3 | 43.1% ± 12.7% | 0.3 ± 0.6   |
| LSTM     | ✓    |        | 3.9 ± 0.5 | 51.5% ± 11.3% | 0.2 ± 0.6   |

- **Narrow**
  - 4.4 ± 0.2: 65.2% ± 4.0%
  - 4.4 ± 0.2: 67.1% ± 3.7%

- **Turtle**
  - 4.5 ± 0.2: 60.3% ± 5.2%
  - 4.5 ± 0.2: 60.2% ± 5.0%

- **Wide**
  - 4.3 ± 0.2: 62.8% ± 3.7%
  - 4.2 ± 0.2: 64.1% ± 3.1%

- **C. Turtle**
  - 4.6 ± 0.2: 57.1% ± 5.8%
  - 4.7 ± 0.2: 59.5% ± 4.9%

E. Discussion

The results highlight some differences between the methods. In BIDA, the model is trained to mimic the augmented data distribution. Although the models can generate original levels, it is likely that the output range will be somewhat similar to the augmented training distribution. This explains why an extension like diversity sampling improves the output diversity by modifying the training distribution to diversify the solutions. On the other side, the training objective of PCGRL directly focuses on generating playable levels regardless of the output distribution. The results show that PCGRL's approach achieves superior playability and diversity. This comes at a higher training time since the agent must learn the game's functional requirements without existing examples. Among the BIDA models, LSTMs seem to achieve the highest playability and diversity. Unlike GANs, VAEs, and VAEGANs, LSTMs do not use convolutions and build the level iteratively, but we leave investigating whether these are the reason for the LSTMs' advantage for future work.

Regarding controllability, the BIDA models treat the controls as inputs to the neural network so it is more likely the output will be unplayable if the control target is far from the values seen in the augmented training data. On the other hand, the PCGRL agent only knows the direction towards the targets, so it has to build a playable level then incrementally update it to reach the targets. So, PCGRL is more likely to generate a playable level regardless of the target values. However, it is hard to plan ahead if the target value is unknown (especially within a limited number of changes), which could be the reason behind PCGRL's relatively higher control error.

IX. Conclusion

In this article, we conducted an experimental study on four recent methods for the Sokoban level generation. We adapted bootstrapping conditional GANs [8] for Sokoban, then applied its methodology to VAES, VAEGANs, and LSTMs. To improve the diversity, we proposed diversity sampling where the model trains on batches containing diverse solutions. To increase the
quality, we proposed training with auxiliary targets and sampling conditions from GMMs. In our experiments, the results showed that uncontrollable PCGRL achieves superior quality and diversity with the main drawbacks being uncontrollability and the long training time. When control is required, PCGRL still exhibits superior quality and diversity, but it is outperformed by LSTMs regarding the control over the generated levels. We also showed that diversity sampling consistently increases the solution diversity. In addition, we showed that training with auxiliary targets can be used to improve the playability rate and that sampling conditions from GMMs increases the probability of generating a playable level.

REFERENCES

[1] K. Compton, “So you want to build a generator...” Feb. 2016. [Online]. Available: https://galaxymate0.tumblr.com/post/139774965871/so-you-want-to-build-a-generator

[2] A. Summerville et al., “Procedural content generation via machine learning (PCGML),” IEEE Trans. Games, vol. 10, no. 3, pp. 257–270, Sep. 2018.

[3] I. Goodfellow et al., “Generative adversarial nets,” in Proc. Adv. Neural Inf. Process. Syst., vol. 27, 2014, pp. 2672–2680. [Online]. Available: https://dl.acm.org/doi/10.5555/2969033.2969125

[4] A. Graves, “Generating sequences with recurrent neural networks,” 2013, arXiv:1308.0850.

[5] A. van den Oord et al., “WaveNet: A generative model for raw audio,” 2016, arXiv:1609.03499.

[6] A. Summerville, “Expanding expressive range: Evaluation methodologies for procedural content generation,” in Proc. 14th Artif. Intell. Interactive Digit. Entertainment Conf., 2018, pp. 116–122.

[7] Y. Murase, H. Matsubara, and Y. Hiraga, “Automatic making of Sokoban problems,” in Proc. Pacific Rim Int. Conf. Artif. Intell., 1996, vol. 1114, pp. 592–600.

[8] R. R. Torrado, A. Khalifa, M. C. Green, N. Justesen, S. Risi, and J. Togelius, “Bootstraping conditional GANs for video game level generation,” in Proc. IEEE Conf. Games, 2020, pp. 41–48.

[9] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, “PCGRL: Procedural content generation via reinforcement learning,” in Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertainment, 2020, pp. 95–101.

[10] S. Earle, M. Edwards, A. Khalifa, P. Bontrager, and J. Togelius, “Learning controllable content generators,” in Proc. IEEE Conf. Games, 2021, pp. 1–9.

[11] P. Bontrager and J. Togelius, “Learning to generate levels from nothing,” in Proc. IEEE Conf. Games, 2021, pp. 1–8.

[12] D. P. Kingma and M. Welling, “Auto-encoding variational Bayes,” in Proc. Int. Conf. Learn. Representations, 2014.

[13] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, “Autoencoding beyond pixels using a learned similarity metric,” in Proc. 33rd Int. Conf. Mach. Learn., vol. 48, 2016, pp. 1558–1566.

[14] L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, “Adversarial reinforcement learning for procedural content generation,” in Proc. IEEE Conf. Games, 2021, pp. 1–8.

[15] N. Shaker et al., “The 2010 Mario AI championship: Level generation track,” IEEE Trans. Comput. Intel. Artif. Intell., 2011.

[16] M. Gudziale and M. O. Riedl, “Game level generation from gameplay videos,” in Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertainment, 2016, pp. 44–50.

[17] C. Salge, M. Green, and R. Canaan, “The AI settlement generation challenge in Minecraft,” Künstl Intell., vol. 34, pp. 19–31, 2020.

[18] G. Smith and J. Whitehead, “Analyzing the expressive range of a level generator,” in Proc. Workshop Procedural Content Gener. Games, 2010, pp. 1–7.

[19] A. J. Summerville and M. Mateas, “Super Mario as a string: Platformer level generation via LSTMs,” in Proc. Digit. Games Res. Assoc./Soc. Advance. Sci. Digit. Games, 2016.

[20] A. Sarkar, Z. Yang, and S. Cooper, “Conditional level generation and game blending,” in Proc. AIIDE Workshop Exp. AI Games, 2020.

[21] A. Odena, C. Olah, and J. Shlens, “Conditional image synthesis with auxiliary classifier GANs,” in Proc. Int. Conf. Mach. Learn., 2017, pp. 2642–2651.

[22] Y. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, “Self-attention generative adversarial networks,” in Proc. 36th Int. Conf. Mach. Learn., 2019, pp. 7354–7363.

[23] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016, pp. 770–778.

[24] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” in Proc. Conf. Neural Inf. Process. Syst. Workshop Deep Learn., Dec. 2014.

[25] T. Tieleman and G. Hinton, “Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude,” COURSERA: Neural Netw. Mach._learn., vol. 4, no. 2, pp. 26–31, 2012.

[26] S. Roberts, D. Husmeier, J. Rezek, and W. Penny, “Bayesian approaches to Gaussian mixture modeling,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 20, no. 11, pp. 1133–1142, Nov. 1998.

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, arXiv:1707.06347.

[28] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in Proc. Int. Conf. Learn. Representations, 2015.

[29] A. Guez et al., “An Investigation of Model-Free Planning,” in Proc. 36th Int. Conf. Mach. Learn., vol. 97, 2019, pp. 2464–2473.