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

Yahia Zakaria, Mayada Hadhoud and Magda Fayek

Abstract—Deep learning for procedural level generation has been explored in many recent works, however, experimental comparisons with previous works are either nonexistent or limited to the works they extend upon. This paper’s goal is to conduct an experimental study on four recent deep learning procedural level generation methods for Sokoban (size = 7×7) to explore their strengths and weaknesses and provide insights for possible research directions. The methods will be bootstrapping conditional generative models, controllable & 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 efficiency and performance. For the bootstrapping method, we propose using diversity sampling to improve the solution diversity, auxiliary targets to enhance the models’ quality and Gaussian mixture models to improve the sample quality. The results show that diversity sampling at least doubles the unique plan count in the generated levels. On average, auxiliary targets increases the quality by 24% and sampling conditions from Gaussian mixture models increases the sample quality by 13%. Overall, PCGRL shows superior quality and diversity while generative adversarial networks exhibit the least control confusion when trained with diversity sampling and auxiliary targets.

Index Terms—Procedural content generation, Level generation, Deep learning, Reinforcement learning, Generative models, Neural networks.

I. INTRODUCTION

PROCEDURAL content generation (PCG) is the process of automatically generating content using algorithms. For decades, PCG was a common tool in the video game and movie industries. It was also employed for enhancing the generalization in machine learning [1]. The generated content can be of an aesthetic nature such as textures, meshes, animations, music and sound effects. It can also be of a functional nature such as levels, enemies, items and even rules. Despite being around for decades, PCG is still notoriously hard. It is usually time and effort consuming to build an efficient generator that yield useful, interesting and diverse content. Even if a generator can generate useful and unique content, it can still suffer from the “10,000 bowls of oatmeal” problem coined by Kate Compton [2]. While every bowl of oatmeal is unique, this uniqueness has little to no effect on the perceived experience so there is no “Perceptual Uniqueness”. We faced a similar issue during our experiments; hence, we will propose a method to mitigate it.

There are different approaches for PCG such as constructive PCG, search for content and search for content generators. An-

other approach is Procedural Content Generation via Machine Learning (PCGML) [3] where models learn to generate the desired content. If the model is a deep neural network in which high-level features are implicitly learned, then the algorithm is classified as Deep Learning. There are generative models, parameterized using deep neural networks, which include Variational Autoencoders (VAE) [4], Generative Adversarial Networks (GAN) [5], VAEGANs where VAEs and GANs are combined. Deep neural networks can also parameterize agents trained via Reinforcement Learning (RL) to construct levels [6]. All the methods, in our experiments, are categorized under PCGML and utilize deep learning.

Given the field’s novelty, it is rare to find an experimental comparison between different methods for multiple reasons. First of all, comparing generators is difficult since it is not possible to distill all the properties of a generator into a single or even a few metrics. There has been some proposed methods and metrics for generator evaluation [7] and many of them are employed in recent works to explore their generators’ properties. Still, they are rarely used to compare the generators with previous works. Another reason is that some papers attend to novel problems that it is not feasible to compare their methods with previous works. Finally, even if it is possible to compare with previous works, they rarely work on a common benchmark (same game, dataset, level size, etc) so their results become incomparable.

The main goal of this paper is to present an experimental comparison between four recent methods that employ deep learning to generate grid-based 2D games. The benchmark will be Sokoban [8] level generation of size 7×7 . The methods will be bootstrapping conditional generative adversarial networks [9], controllable & non-controllable procedural content generation via reinforcement learning (PCGRL) [6], [10] and generative playing networks (GPN) [11]. Throughout the paper, we will discuss the difficulties we faced while adapting the four methods to the benchmark and we will present the modifications added to either adapt the methods, enhance their properties or improve their efficiency. Thus the contributions of this paper will be as follows:

- 1) Adapt the bootstrapping method [9] (with GANs, VAEs and VAEGANs) and Generative Playing Networks [11] to Sokoban level generation.
- 2) Propose Diversity Sampling to improve the generators ability to generate levels with diverse solutions.
- 3) Propose using Auxiliary Targets to enhance the generators quality.
- 4) Propose sampling conditions from a Gaussian Mixture Model to improve the quality of the generated levels.

- 5) Compare between the four methods based on the experimental results.

In section II, we will mention recent related works. Then we will formulate a problem definition for the level generation task, state the objectives and explain the rules of Sokoban (the benchmark game for our experiments) in section III. In section IV, we briefly explain the bootstrapping method and present our modifications to improve the method's quality and diversity. Section V briefly explains PCGRL and discusses a simple trick to significantly decrease the training time. In section VI, we briefly explain the generative playing networks then we present a simple modification to the reward function so that it can be applied to level generation for puzzle games. Then we will detail the experimental setup in section VII and we will show and discuss the results in section VIII. Finally, we will present the conclusion in IX and wrap up with some of the planned future works in section X.

II. RELATED WORKS

There are multiple recent works that review the PCG methods with different focus scopes. Procedural Content Generation via Machine Learning (PCGML) [3] was reviewed where various learning methods in addition to different data sources and representations were discussed. In a more recent work, Deep learning for procedural content generation [12] was reviewed and the review also discussed deep learning methods that are potentially useful for PCG but were still not widely used. Another recent work [13] presented a review of PCG with a focus on puzzle generation. These reviews include a large variety of methods which cover a wide range of applications and games. Our work differs in the fact that we focus on a smaller set of recent works but with the goals of presenting an experimental study rather than a literature review.

There is a variety of generative models that rely on deep learning such as Variational Autoencoders (VAE) [4], Generative Adversarial Networks (GANs) [5], Auto-Regressive Models [14] and more. Many of these models has already been applied to PCG such as using GANs to generate Zelda levels [9], using VAEs for generating and blending levels from multiple games [15] and using LSTMs to generate Sokoban Levels [16]. All of the aforementioned methods rely on training the model to capture the distribution of the training data. Thus they usually fail to generalize when presented with small datasets which is unfortunately the case for most games during development. The bootstrapping method [9] address this issue by searching the model's output for playable levels that can be used to train the model in the upcoming iterations. The method was applied with Self-Attention GANs [17] but they are also applicable to other generative models. Another contribution alongside the bootstrapping method was the conditional embedding [9] which was used to improve the generator's quality and diversity. In this paper, we will also show that conditional embedding can be used to control the generator output. Generative Playing Networks (GPN) [11] showed that a generator can be trained to generate levels from scratch using feedback from an agent trained to play the game.

The generator is trained using the gradients back-propagated from a critic network with the goal of finding winnable but hard levels. To train without data, the agent is trained on the output of the generator while the generator is trained using the agent; thus building a closed loop where both the agent and the generator improves at their own tasks in tandem. The training process can be bootstrapped using an initial set of levels to train the playing agent.

Another approach for PCG is to train an agent to build a level. Procedural Content Generation via Reinforcement Learning (PCGRL) [6] trains an agent to modify tiles in the level where it is rewarded if the level's quality improves. The work was extended [10] to include controls that allow the user to specify the desired properties of the generated level. Both methods rely on the reward function to discriminate between low and high quality levels. Adversarial Reinforcement Learning for PCG (ARLPCG) also trains a generator agent, however, the reward depends on the performance of a playing agent trained to win the generated levels. It was tested on 3D platforming and racing so we did not include it in our study since we still need more work to adapt it for Sokoban.

III. PROBLEM DEFINITION

To lay the foundation for the methodology, we will present an abstract formulation of the problem followed by a set of objective we seek to optimize. Finally, we will define the rules of Sokoban and the functional requirements for valid Sokoban levels.

A. Level Generation

The procedural level generation task can be abstracted as a function that takes a random sample from the domain and return a point from the level space 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. In case of conditional generators, a condition u is supplied to the generator as shown in 2.

$$l = G(z) \quad (1)$$

$$l = G(z|u) \quad (2)$$

Equation 1 covers PCGRL and GPN since they are not controllable while equation 2 covers controllable PCGRL and the other generative models covered in the methods section. For both versions of PCGRL, the generator's domain is the level space since the agent is supplied a level (usually contains random tiles). For GPN and the other generative models, the generator's domain is the latent space where the input is sampled from a standard normal distribution. PCGRL also comes with a nice feature where we could supply the generator with a limit on the number of changes which was shown to be useful for mixed-initiative level design [18].

B. Objectives

In most cases, generators cannot be evaluated using a single objective. A good generator should exhibit high quality and high diversity and if it is a controllable generator, it should

also exhibit high controllability. In some cases, compromises must be made, since optimizing one objective can lead to degradation in other objectives. In this section, we will discuss the different objectives that we seek to optimize.

1) *Quality*: In our case, measuring the generators' quality is relatively simple since the functional requirements for a game level is verifiable using a solver program. In our experiments, we will use the percentage of playable levels from a generated sample to be the measure of a generator's quality.

2) *Diversity*: A generator that return the same playable level for every input would receive a perfect quality score. However, this lack of diversity would render it useless. Unlike quality, it is not trivial to quantify a generator's diversity. In [9], the lack of diversity was measured as the percentage of duplicate levels in the generated sample. In another paper [10], diversity was measured as the average tile-wise hamming distance between all pairs of levels in the generated sample. Although we will include both of these measures in the results, we argue that both of them are flawed since most of the tiles have no impact on how the level is solved. For example, in Fig.1, all the tiles on the levels' borders are insignificant and can be partially or fully removed without affecting the levels' solutions. These tiles are almost half the level (24 out of 49 tiles), so we can create 2^{24} different variations of the level just by removing some of these walls. On the other hand, the solution could drastically change by modifying just a single tile.

Fig. 1. Examples of Sokoban Levels

A better measure would be the number of unique solutions in the generated sample where a solution is defined as a sequence of actions to reach a goal state. However, we argue that levels could have unique solutions without being perceptually unique. For example, all the levels in Fig.1 have different solutions despite boiling down to the same high-level plan: go to the tile from which you can push the crate towards the goal (if needed) then keep pushing till it reaches the goal. We will discuss another option that we argue to be an improvement over solutions while presenting diversity sampling in section IV-B.

Quantifying the generator's diversity as single number is usually not enough since it is desirable to achieve diversity over more than a single property. Thus, we will follow recent works [10] and visualize the expressive range of the generator over the solution length and the number of crates.

3) *Controllability*: The goal of controllable generators is to design levels whose properties are as close as possible to the supplied targets. A simple quantification of controllability is the mean error between the target and actual properties of

the generated sample. Confusion matrices will also be used in order to visualize the control error for different target values.

C. Sokoban

In this paper, we will conduct our experiments on the task of level generation for the Sokoban game. Sokoban [8] is a 2D grid-based puzzle game with discrete actions and deterministic state transitions. Sokoban is known for being an unforgiving game where actions, that lead to dead ends, are common. Therefore Sokoban is hard for reinforcement learning agents due to its reliance on long-term planning. In addition, the state space is usually huge so it is inefficient to solve via searching without using domain knowledge to prune dead-end states as early as possible.

The rules of Sokoban can be described as follows:

- All the game states must satisfy the following requirements:
 - There is one player only (no more, no less).
 - The number of crates are equal to the number of goals.
 - A tile can be in one of the 7 states shown in TABLE I. Hence, only a goal can coexist with another object which could be either a crate or a player.
- The actions are moving left, right, up or down.
- The actions affect the states as follows:
 - The player will attempt to move to the adjacent tile in the action's direction.
 - If the tile does not contain a crate or a wall, the action succeeds.
 - If the tile contains a wall, the action fails.
 - If the tile contains a crate, the crate will attempt to move one step in the action's direction. If the crate's destination contains a wall or another crate, the action will fail and nothing will move. Otherwise, the action succeeds and both the player and crate move to their destination.
- A state would be a goal state (player wins) if every crate is on a tile containing a goal and vice versa.

TABLE I
SOKOBAN TILESET

Tile Content	Sprite
Empty	
Wall	
Goal	
Crate	
Player	
Crate & Goal	
Player & Goal	

We consider a Sokoban level to be compilable if it satisfies the requirements to be considered a Sokoban game state. But

we add another condition that 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 program we used for our experiments. The generative models and GPN will take into consideration all the 7 possible tile states shown in TABLE I, while only the first 5 tiles are used by PCGRL [6] and Controllable PCGRL [10].

IV. BOOTSTRAPPING GENERATIVE MODELS

Generative models are trained to capture the probability distribution of the training data. So when the data is scarce, the models will be prone to overfitting and will mostly generate copies from the training data. This is almost the case with game level datasets, especially for games under development. To mitigate this issue, a bootstrapping method was proposed [9] to augment the training set with any new samples generated by the model that satisfy the functional requirements of the game. This method exploits the fact that functional validity of game levels are usually verifiable programmatically. The augmentation process is applied multiple times during the training process so the final version of the model is trained on both the original dataset and all the valid levels generated by its previous versions.

Alongside the bootstrapping method, conditional embedding self-attention generative adversarial networks (CESAGAN) was proposed [9]. The self-attention module enhances the model's ability to capture long distance relationships between tiles. The conditional embedding enhances the model's ability to respect the playability requirements related to the tile frequencies in the generated level. The model was trained to generate level for the GVGAI version of the Zelda game [19] and the conditional embedding were derived from the frequency of each tile type in the level. To adapt conditional embeddings for Sokoban, we use different inputs:

- 1) **Walls**: which is the wall count divided by the level's area.
- 2) **Crates**: which is the crate count divided by the level's area.
- 3) **Solution Length**: which is the minimum number of actions required to solve the problem divided by the level's area.
- 4) **Player distance from the center**: which is the Manhattan distance between the player and the level's center. The player location on each axis is normalized to be in the range $[-1, 1]$ where 0 is the level's center on the corresponding axis.

Initially, the walls and crates were picked to match how the tile frequencies were used to form the condition in the original paper [9]. Later, we added the solution length since we were interested in finding if the difficulty of the generated level can be controlled via the conditional embedding. The player distance from the center was finally added to resolve an issue noticed during the experiments where player was always located at a certain position in all the generated levels. The player distance was used instead of the xy-coordinates to prepare for using data augmentation where we would flip

and rotate the levels since the distance is invariant to these operations. Although flipping and rotation was not used for data augmentation in the experiments, the player distance was not replaced by the xy-coordinates since it was serving its purpose while being only one feature instead of two.

A. Generative Models

We test the bootstrapping method with three types of generative models. While the bootstrapping method was originally applied to GANs only, they can be applied to VAEs and VAEGANs without any modifications, thus we added them to our experiments.

1) *Generative Adversarial Networks*: In Generative Adversarial Networks (GAN) [5], two networks are trained in an adversarial manner. The discriminator is trained to discriminate between real and fake data. On the other hand, the generator is trained to generate fake data that the discriminator cannot differentiate from real data. The generator utilizes the gradients returned from the discriminator to optimize its own weights. Ideally, the generator will generate fake data from the same distribution as the real data and the discriminator will no longer be able to differentiate between the real and fake samples. In our experiments, we use the hinge version of the adversarial loss [20] as shown in (3) and (4) which was used in the original paper of the bootstrapping method [9].

$$L_D = \mathbb{E}_{x \sim P_x} [\max(0, 1 - D(x|u))] + \mathbb{E}_{z \sim \mathcal{N}(0, I)} [\max(0, 1 + D(G(z|u)|u))] \quad (3)$$

$$L_G = -\mathbb{E}_{z \sim \mathcal{N}(0, I)} [D(G(z|u)|u)] \quad (4)$$

L_D and L_G are the loss functions of the discriminator D and generator G respectively. The real data x is 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.

2) *Variational Autoencoders*: As an extension to autoencoders, Variational Autoencoders (VAE) [4] contains two networks: an encoder that compresses the given data point to a latent vector and a decoder that reconstructs the original data point from the latent vector. However, the encoder in VAEs outputs the parameters of a Gaussian distribution (the mean and the logarithm of the variance) in the latent space. The training objective is to optimize the evidence lower bound loss as shown in (5).

$$L_E = L_G = -\mathbb{E}_{x \sim P_x} [\mathcal{H}(x, G(z|u)) + D_{KL}(E(x|u) || \mathcal{N}(0, I))] , \quad z \sim E(x|u) \quad (5)$$

To avoid confusion with the discriminator (which is denoted by D), we will use G to denote the decoder since it is used as a generator. The same loss L_E & L_G is applied to the encoder E and the decoder G , however, it is noteworthy that Kullback–Leibler divergence term D_{KL} only affects the encoder. The latent vector z is sampled from the distribution predicted by the encoder using the reparameterization trick [4]. The reconstruction term uses the cross-entropy loss $\mathcal{H}$ over the tiles.

3) *Variational Autoencoder Generative Adversarial Networks*: It is possible to train a generative model as a generator in a GAN and a decoder in a VAE. This model is called Variational Autoencoder Generative Adversarial Networks (VAEGAN) [21]. The motivation behind this model was to replace the pixel-wise reconstruction loss with an alternative discriminator-based loss function which should incentivize the VAE to find a reconstruction that is perceptually similar rather than optimizing the pixel-wise loss. The same motivation could be of interest for level generation where the generator is trained to generate a level that has similar features compared to the original level rather than seeking to copy it tile-wise. The loss functions of the VAEGAN combines the losses of GANs and VAEs with the reconstruction loss being changed from cross-entropy over the tiles to the mean-squared error over a feature vector extracted from the discriminator’s last hidden layer as shown in (6), (7) and (8).

$$\begin{aligned} L_D = & \mathbb{E}_{x \sim P_x} [\max(0, 1 - D(x|u))] \\ & + 0.5 \mathbb{E}_{x \sim P_x} [\max(0, 1 + D(G(z_r|u)|u))] \\ & + 0.5 \mathbb{E}_{z_n \sim \mathcal{N}(0, I)} [\max(0, 1 + D(G(z_n|u)|u)), \\ & z_r \sim E(x|u) \end{aligned} \quad (6)$$

$$\begin{aligned} L_G = & 0.5 \mathbb{E}_{z \sim \mathcal{N}(0, 1)} [-D(G(z|u)|u)] \\ & + 0.5 \mathbb{E}_{x \sim P_x} [-D(G(z_r|u)|u)] \\ & + \alpha \|h_D(x) - h_D(G(z_r|u))\|^2, \quad z_r \sim E(x|u) \end{aligned} \quad (7)$$

$$\begin{aligned} L_E = & -\mathbb{E}_{x \sim P_x} [\|h_D(x) - h_D(G(z_r|u))\|^2] \\ & + D_{KL}(E(x|u) \parallel \mathcal{N}(0, I)), \quad z_r \sim E(x|u) \end{aligned} \quad (8)$$

The feature vector from the last hidden layer in the discriminator is denoted by h_D . The parameter α controls the weight of the mean-square error loss in the generator loss.

B. Diversity Sampling

The bootstrapping method relies on the output of the semi-trained generator to augment the dataset with more samples for training in the upcoming iterations. This means that the output distribution of the generator during the first few augmentation runs will dictate the distribution of the training data for the upcoming iterations. This is worrisome given how unstable GAN training is. GANs are prone to mode collapse and if the generator only learns a subset of the modes, these modes will receive more levels and will overrun the other modes.

Our first attempt to mitigate this issue was by changing how we sample the conditions. Initially, the conditions were sampled from the training data while training the GAN networks and while sampling new levels from the generator to augment the dataset. We kept using this sampling process for training the discriminator. For training the generator and for the dataset augmentation, we switched to sampling the conditions uniformly from the range bound by the minimum and maximum values found in the dataset. Thus the dataset is augmented with new levels with a diverse set of conditions even if the original condition distribution was focused around a few modes. However, the generator still created levels that

have very similar solutions. After all, the generator can satisfy the rate condition by adding more crates that are already on the goal. It can also vary the solution length by changing the player’s placement and, sometimes, by adding walls to lengthen the path to the first push. Thus we noticed that many levels are different condition-wise but have similar solutions.

Therefore, we propose diversity-sampling based on the level solution to improve the diversity of the generators output. Using cluster-based sampling, we group the training data into clusters where levels in the same cluster are close to each other solution-wise. Instead of sampling a real level uniformly, we randomly pick a cluster, then sample a level from it. Thus a level is less likely to be sampled if there are many levels in the dataset with similar solutions. The motivation is ensure that the level distribution used to train the model is not biased towards a certain solution even if it overruns the dataset. This should help diversify the solutions found by the data augmentation process and improve the training dataset solution-diversity.

However, how can we decide if two solutions are closely similar or not? Sequence distance measures, such as edit distance, poorly model the perceptual difference between levels. In the end, we decided to use a distilled version of the solution which we will call a “Signature”. It is computed as shown in Algorithm IV-B where the input is the solution represented as a string of actions which can be moves (denoted by lower case directions $\{r, l, d, u\}$) or pushes (denoted by upper case directions $\{R, L, D, U\}$). It works by keeping only the first action of each consecutive sequence of the same action, then removing all the non-push actions. To make the signature rotation and flipping invariant, we rotate the signature till the first action is “R” and after that, if there are any vertical actions and the first vertical action is “D”, we flip the signature along the y-axis.

The intuition behind the signature representation is to view the solution as a sequence of high-level actions (HLA) instead of primitive actions. Humans tend to search for solutions in a hierarchical manner where the plan is formulated in terms of HLAs. Refining the plan to primitive actions usually requires little to no mental effort, so it should have no effect on the perceptual uniqueness of the plan. Based on how we plan in Sokoban, we can construct the plan using the HLA “Push(C, D, N)” where C is the crate to be pushed, D is the direction in which it will be pushed and N is the number of pushes. The refinement of this HLA would contain a sequence of moves to reach the crate followed by N pushes in the direction D . The signature (before rotating and flipping it) is just a version of the high-level plan where the crate C and the number of pushes N are ignored. We ignore these two arguments such we can use a naive clustering method where each cluster contains only levels with the same signature; thus no distance function is needed. While this is probably not the best solution, it was enough to significantly increase the number of unique solutions generated by the models. Searching for better options will be left for future work.

C. Auxiliary Targets

A Sokoban level is compilable if and only if the number of goals is equal to the number of crates. In our early GAN

Algorithm 1 Extract Signature from Solution

```

procedure EXTRACTSIGNATURE(actions)
  signature  $\leftarrow []$ 
  if actions[0] is a push then
    append actions[0] to signature
  end if
  for index from 1 to length(actions) - 1 do
    if actions[index] is a push then
      if actions[index]  $\neq$  actions[index - 1] then
        append actions[index] to signature
      end if
    end if
  end for
  while signature[0]  $\neq$  'R' do
    rotate signature by 90°
  end while
  for index from 1 to length(signature) - 1 do
    if signature[index] = 'U' then
      break
    else if signature[index] = 'D' then
      flip signature along y-axis
      break
    end if
  end for
  return signature
end procedure

```

experiments, we did not strictly enforce this constraint and accepted levels where the number of goals exceed the number of crates. We noticed that the generator was struggling to generate levels that satisfy the equality constraint. It seems that the discriminator was less successful at implicitly learning the equality constraint through training to differentiate between real and fake data. To facilitate the training process, we added an extra target to the discriminator (similar to the Auxiliary Classifier GAN [22]) where it has to differentiate between compilable and uncompileable levels. In addition, we add regression targets where it has to count the objects in the level (walls, players, crates and goals) since they are useful for deciding whether the level is compilable or not. This extension proved useful to enhance the quality of the generators.

D. Sampling Conditions from a Gaussian Mixture Model

All of the three generative models, we discussed in section IV-A, are conditional. So to generate a level, we need to supply the generator with a latent vector and a condition. Latent vectors are sampled from a standard normal distribution, however, conditions do not have a parameterized distribution to sample from. It may be argued that conditions are meant to be controlled and not to be randomly sampled, however, it is desirable to give the user control over what conditions they want to control (if any) and what conditions they do not care about.

The first option is to sample the condition from a uniform distribution bounded by the minimum and maximum condition values extracted from the final training dataset. However, a

large region of the space in the condition hypercube could be invalid. For example, there is an upper limit on the solution length for a level containing a single crate. Since the generator could not have possibly seen a condition where the number of crates is low (e.g. 1) while the solution length is high (e.g. 90 steps) in the training data, it is reasonable to expect bad quality outputs in these regions.

The second option is to sample the conditions from the final dataset. However, there are a few drawbacks:

- 1) The final dataset contains thousands of levels and it is susceptible to increase if the model is trained longer. So it may not be data efficient to ship all the seen conditions with the generator.
- 2) There is no easy method to conditionally sample from a list of conditions especially if the query condition values are nonexistent in the final dataset.
- 3) The generator will always sample from previously seen conditions which could possibly limit the output diversity.

Therefore, we propose fitting the conditions in the final training dataset into a Gaussian mixture model. It is much more compact than storing the full set of conditions and can be sampled unconditionally as shown in (9) and conditionally as shown in algorithm IV-D where the query variables are b with value x_b and the unknown variables are a . The Gaussian mixture model is specified by n normal distributions $\mathcal{N}(\mu_k, \Sigma_k)$ where the weight of each distribution in the mixture is w_k .

$$u \sim \mathcal{N}(\mu_k, \Sigma_k), k \sim [1, n], P(k) = w_k \quad (9)$$

Algorithm 2 Conditionally Sample from a Gaussian Mixture Model

```

procedure CONDITIONALSAMPLE(xb,  $\mu_{1..n}$ ,  $\Sigma_{1..n}$ , w1..n)
  for k from 1 to n do
     $\begin{bmatrix} \Sigma_{k_{aa}} & \Sigma_{k_{ab}} \\ \Sigma_{k_{ba}} & \Sigma_{k_{bb}} \end{bmatrix} \leftarrow \Sigma_k$ 
     $\begin{bmatrix} \mu_{k_a} \\ \mu_{k_b} \end{bmatrix} \leftarrow \mu_k$ 
     $\bar{\mu}_k \leftarrow \mu_{k_a} + \Sigma_{k_{ab}} \Sigma_{k_{bb}}^{-1} (x_b - \mu_{k_b})$ 
     $\bar{\Sigma}_k \leftarrow \Sigma_{k_{aa}} - \Sigma_{k_{ab}} \Sigma_{k_{bb}}^{-1} \Sigma_{k_{ba}}$ 
     $\hat{w}_k \leftarrow \frac{w_k}{\sqrt{|\Sigma_{k_{bb}}|}} e^{-\frac{1}{2} (x_b - \mu_{k_b})^\top \Sigma_{k_{bb}}^{-1} (x_b - \mu_{k_b})}$ 
  end for
  sample  $k \sim [1, n]$  where  $P(k) = \frac{\hat{w}_k}{\sum_{i=1}^n \hat{w}_i}$ 
  sample  $u_a \sim \mathcal{N}(\bar{\mu}_k, \bar{\Sigma}_k)$ 
  return  $\begin{bmatrix} u_a \\ x_b \end{bmatrix}$ 
end procedure

```

V. PROCEDURAL CONTENT GENERATION VIA REINFORCEMENT LEARNING

PCGRL [6] formulates the level generation problem as a Markov decision process (MDP) where an agent can act on the level by applying modifications and the reward is

proportional to the improvement in the level quality. A PCGRL Environment has multiple representations out of which we will focus on only the following three:

- 1) **Narrow:** The agent can only modify the tile at its current location but it does not control its own movement. After each action, the environment moves the agent to the next location which can be either selected sequentially or randomly. In our experiments, we configure the environment to move the agent sequentially across the map. In this representation, the observation is centered around the agent so the agent does not need a separate input to identify its position.
- 2) **Turtle:** The agent can only modify the tile at its current location but it can instead choose to move to an adjacent tile. Similar to the narrow representation, the observation is centered around the agent.
- 3) **Wide:** The agent can modify any tile at any location in the level. Therefore, the agent does not need to have a location, thus the observation is not centered.

Controllable PCGRL 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 Sokoban, the controllable properties are the number of crates and the solution length, so the observation will include two extra channels.

For Sokoban, the PCGRL official repository provides a reward function that rewards the agent for making progress towards a compilable playable level that follows a set of constraints. These constraints include having no more than 3 crates and no more than a single empty region. Since we run our experiments on levels of size 7×7 , we increase the limit on the crates up to 5. It is important to note that agent can still create levels with more crates than the crate limit but at a penalty. In addition, the environment rewards the agent for creating levels with longer solutions.

The reward function used in the controllable PCGRL repository is almost the same as its uncontrollable counterpart except that the agent is rewarded for decreasing the gap between the actual and the requested level properties. So there is no need for the crate limit since it was replaced by the crate target. And the agent is no longer rewarded for creating levels with longer solutions since it is now incentivized to achieve the solution length target.

To maintain diversity, PCGRL limits the number of allowed changes. Otherwise, the agent would disregard the initial level and memorize a single level that maximizes the reward. For Sokoban, the default limit was set to 20% of the level map ($20\% \times 5 \times 5 = 5$ tiles). On 7×7 maps, 20% was too low and the agent was unable to achieve any satisfying results. Therefore, we increased the limit to 40% which amounts to 20 tiles. There is an overall limit on the number of steps which is equal to the product of the maximum number of changes and the map area. So for our configuration, the limit was $20 \times 49 = 980$ steps. The final termination condition is to achieve a level whose solution contains at least 18 steps. We initially increased that limit to 100 steps but latter, we removed the limit altogether for reasons that we will discuss in section V-A. For controllable PCGRL, the default change percentage was

set to 100% which we did not change. Controllable PCGRL has an additional termination condition where the episode ends as soon as the targets are achieved.

A. Reducing the Training Time

The bottleneck in the training process of a PCGRL agent for Sokoban is the reward calculation. A factor in the reward is the solution length which can only be calculated by actually solving the level (if possible). The search space of many Sokoban levels is huge, therefore, it can take a long time to find a solution. The original PCGRL repository addresses this issue dividing the search process into 4 stages where the first stage is optimal and the remaining stages trade optimality for speed. Each of the four stages is heavily limited to run for 5000 iterations only. This limit was increased to 10,000 in the controllable PCGRL repository. Unfortunately, we had to increase this limit to double the value for 7×7 levels to be 10,000 and 20,000 in uncontrollable and controllable PCGRL respectively. Add to this the fact that it is more likely for 7×7 levels to exceed the iteration limit at more stages so training for such size was impractical at the current performance.

The first change we tried was to cache the solutions (local to each episode) since it was noted that agents trained on Turtle and Narrow representation spend most of the time moving rather than applying changes. On some small experiments, the training time was almost halved by caching but it was still too slow to run for 100 million steps.

The next thing we noticed is that the CPU utilization was low. We concluded that most of the processor cores spend their time in an idle state while waiting for a few environments to finish their search. So the second change was to run all the environments on the main process and to solve the levels asynchronously on separate processes. The idea is that the environment in uncontrollable PCGRL does not need to wait for the solution since the reward is not needed till the model training phase starts. To be more accurate, it will no longer be needed if we remove the termination condition that relies on the level's solution length; hence, we removed it. Now each environment would submit their level to a solver from a multiprocessing pool and keep a future object. Before each training phase, the main thread is halted till all the solvers finish their search and then the main thread collects the rewards from the future objects. This change decreased the training time from almost 9 days to just 2 days and 8 hours on our machine.

Unfortunately, this does not work for controllable PCGRL since the solution is immediately required before the next observation is sent to the agent. Thus, the only remaining option was to rewrite the solver using the C programming language. Fortunately, the speed gain was more than $400\times$, so, for the same iteration limits, the reward calculation was no longer a bottleneck. We used the C solver in our experiments with controllable PCGRL only since the other PCGRL experiments were already done. While we plan to use the C solver for most of our future experiments with Sokoban, we consider the idea of solving levels asynchronously a useful trick to use if we work with a game that has no fast solvers.

VI. GENERATIVE PLAYING NETWORKS

GPNs [11] are most similar to GANs with the main difference being the replacement of the discriminator with the critic network of a reinforcement learning agent. The reinforcement learning agent is trained to win the game in the least number of steps while the generator is trained to create levels that the agent can win but after as many steps as possible. In other words, the agent seeks to maximize the utility where the reward is defined as shown in (10) while the generator seeks to keep the utility as near to zero as possible. t_n is the episode length, t_{max} is the maximum allowed episode length and r is the original reward returned by the GVGA environment (given for killing enemies, collecting the key and so on). GPNs employ many tricks to facilitate their training. First of all, while the agent is being trained to solve the latest batch of levels, the agent and the generator are trained as an Autoencoder where the agent encodes the state into a latent vector from which the generator reconstructs the original state. Another trick is the use of elitism to keep around the levels, that agent is currently struggling to solve, so that it can continue training on them. Although GPNs can be trained without any initial dataset, the process could be bootstrapped using a few levels to pretrain the agent. In addition, GPNs replace the discount factor with the actor's confidence to combat vanishing reward in the long episodes of Zelda games (up to 1000 steps). To prevent the generator from collapsing to a single level, GPNs use diversity updates where the generator is trained to maximize the distance between the latent vectors (extracted from the agent) of the levels in a generated batch.

$$R_{action}(S_n) = \begin{cases} 1 - \frac{t_n}{t_{max}} & \text{if player wins} \\ -1 + \frac{t_n}{t_{max}} & \text{if player loses} \\ \frac{1}{t_{max}} & r > 0 \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

This method was used to generate Zelda levels only but most of it is applicable to Sokoban without modification. The only necessary change was to replace the reward function of the agent. In the GPN repository, the paper's configuration used the reward function (10) which is suitable for action games since the agent is rewarded for how long they survived in case of losing. However, in Sokoban, the agent loses if and only if the puzzle is not solved before the episode ends. So the best option for the generator is to create unsolvable levels since the utility will be zero. To change that behavior, we modified the reward function as shown in (11) such that the reward for losing is -1 with no regard for the episode length.

$$R_{puzzle}(S_n) = \begin{cases} 1 - \frac{t_n}{t_{max}} & \text{if player wins} \\ -1 & \text{if player loses} \\ \frac{1}{t_{max}} & r > 0 \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

VII. EXPERIMENTAL SETUP

In this section, we will start by introducing the dataset used in our experiments. Then we will present the network architectures for the generative models and mention the changes in

the architectures used by the other methods. Finally, we will detail the rest of the training and generation configuration.

A. Dataset

The dataset shown in Fig.2 was used for bootstrapping the generative models and pre-training the GPN. It contains 12 levels with varying degrees of difficulty. We made sure to include a few trivial levels to help the reinforcement learning agent in the GPN method get a few positive rewards as early as possible. When the dataset is used for bootstrapping generative models, we expand the dataset by solving every level and adding the intermediate states along the solution path into the dataset. The motivation for this expansion is to provide levels with varying values along the solution length condition. The expanded dataset contains 167 levels with 145 unique solution and 16 unique signatures. The maximum solution length found in the dataset is 31 steps. The distribution of the expanded dataset is shown in Fig.3 where it shows how much of the dataset covers each bin in the behavior space.

Fig. 2. Sokoban Dataset

B. Network Architecture

For the bootstrapping method, we use the same generator architecture for all three generative models. The encoders and discriminators are also the same except for the last layer where that discriminator outputs a single value (plus auxiliary targets if enabled) while the encoder outputs the mean and log-variance of a Gaussian distribution in the latent space.

The architecture utilizes self-attention and instance normalization (with learnable affine parameters) in the generator,

Fig. 3. Dataset Distribution

encoder and discriminator. In addition, we use spectral normalization on every convolution layer in the discriminator. A skip connection and a trainable parameter γ was added to the self-attention module as shown in Fig.4 to match the self-attention GAN paper [17]. All the internal activation functions are leaky ReLUs with the negative slope set to 0.2.

The four conditional features are treated similarly in all the three networks where it is passed through two feed-forward layers (16 neurons then 32 neurons) with leaky ReLU activations. Thus the conditional embedding that is supplied to the rest of the network is of size 32. For the generator, the conditional embedding is concatenated with the latent vector and supplied to all the self-attention modules. For the encoder and discriminator, it is expanded to the level size and concatenated to the level tile map, in addition to being supplied to all the self-attention modules. Since the GAN training process is very sensitive to the network architecture, the architectures were tuned for GANs then used for VAE and VAEGAN as-is with no further tuning required.

The generator architecture is as follows (excluding the conditional embedding network):

- 1) Convolution Transpose 5×5 (channels: 64) followed by 2D instance normalization and leaky ReLU.
- 2) Convolution 3×3 (channels: 64, padding: 1) followed by 2D instance normalization and leaky ReLU.
- 3) Self-attention layer with key/query size of 16.
- 4) Convolution Transpose 3×3 (channels: 32) followed by 2D instance normalization and leaky ReLU.
- 5) Convolution 3×3 (channels: 32, padding: 1) followed by 2D instance normalization and leaky ReLU.
- 6) Self-attention layer with key/query size of 8.
- 7) Convolution 3×3 (channels: number of tile types, padding: 1) followed by softmax.

The encoder and discriminator architectures are as follows (excluding the conditional embedding network and the last linear layer):

- 1) Convolution 3×3 (channels: number of tile types, padding: 1) followed by Leaky ReLU.
- 2) Self-attention layer with key/query size of 8.

- 3) Convolution 3×3 (channels: 32, padding: 1) followed by 2D instance normalization and leaky ReLU.
- 4) Convolution 3×3 (channels: 64) followed by 2D instance normalization and leaky ReLU.
- 5) Self-attention layer with key/query size of 16.
- 6) Convolution 3×3 (channels: 64, padding: 1) followed by 2D instance normalization and leaky ReLU.
- 7) Convolution 5×5 (channels: 64) followed by leaky ReLU.

For PCGRL, We follow the agent architectures found in the official repository where we only added one more convolution layer 3×3 and one more ReLU activation in each network. We also use the proximal policy optimization algorithm with the same configuration except that we used two separate networks: one for the actor and another for the critic, since we found that it improved the training process.

For GPN, the agent network and the generator network had to be redesigned to fit a 7×7 level. Following the model used in the repository, we used a Residual Network [23] followed by a single GRU unit [24] for the agent. The GRU unit was required for Zelda since it is partially-observable (there no observations for the player direction, last enemy move frames, etc) so the agent had to use a recurrent policy. For Sokoban, a recurrent policy is no longer necessary but removing the GRU unit lead to weaker performance so we kept it. The generator uses Convolution Transpose (3×3), Batch Normalization, ReLU and Dropout.

C. Training and Generation Configuration

For the bootstrapping method, we run the training for 10,000 iterations with 32 batch size. Every 100 iterations, the generator is used to create 128 levels out of which new playable levels are added to the training data. The RMSProp optimizer [25] was used with the learning rate set to 0.001 and the smoothing constant set to 0.9. For VAEGAN, the factor of the mean square loss α is 0.01. When auxiliary targets are enabled, the regression losses are weighted by 1/49, 1/5, 1/5, 1 for the walls, goals, crates and players. All the regression losses as L1 and the compilability target loss is binary cross entropy. For fitting the condition to a parameterized distribution, we used a Bayesian Gaussian mixture model [26] with 16 components, each with its own separate full covariance matrix.

To verify levels, we use a breadth-first search agent with a limit of 10 million search iterations. To improve the search performance, we prune states where the pushed crate enters an empty tile from which it can no longer reach a goal. The same solver with the same limit is used to verify the outputs of all the methods to extract the results.

Since some of the generative were unstable, we ran each experiments 5 times. To save space, the presented results show the mean of the 5 runs and we will report any significant variations in the results for each model and configuration.

For uncontrollable PCGRL, we run each experiment once for 100 million steps divided on 50 parallel environments. For controllable PCGRL, the agent was trained for 500 million steps. The agent was trained using the Proximal Policy Optimization (PPO) algorithm [27] with the Adam optimizer [28]

Fig. 4. Self-Attention Module

with the learning rate set to 0.0001 and the gradient norm clipped at 0.5. The agent’s networks is trained after every 128 steps from each environment where the agent is optimized on the collected experience 4 times. These parameters follow the original experiments [6], [10] done for 5×5 Sokoban level generation. The change percentage for uncontrollable PCGRL was changed from 20% to 40% while for controllable PCGRL, it was left at 100%. During generation, the change percentage was set to 100% for both. The maximum number of crates for uncontrollable PCGRL was set to 5 crates. For controllable PCGRL, the targets range was to $[1, 10]$ for the crates and $[1, 105]$ for the solutions. The crate target range was originally $[1, 17]$ but the agent performance was poor since it is hard to learn how to make solvable levels with a large number of crates. For each agent, the generation process is run twice where the actor network is treated as a deterministic policy in one run and as a stochastic policy in the other. When the actor is considered deterministic, the episode was terminated as soon as a previously-seen observation is encountered as it means that the agent is stuck in a loop till the episode ends.

For GPN, we use the Sokoban dataset to pre-train the agent for 20 million steps divided over 64 environments. The episode length was limited to 50 steps since all the levels used for pretraining can be solved in 31 steps or less. Since the agent was struggling to solve any nontrivial levels, we did not find it useful to increase the step limit. To improve the agent’s generalization, we augment the dataset by flipping and rotating the levels. Following the original configuration from the repository, the generator is trained for 100 updates where in each update, it is trained to optimize the level utility for 10 iterations, then it is trained to increase the level diversity for 90 iterations. The generator’s batch size is 128 and the latent size is 512. After each update the agent is trained for 1 million steps on environments that have a 50% chance of being sampled from the elites and 50% chance of being sampled from the generated levels. The elites are selected based on the

maximum rewards achieved by the agent in the given level. We did not attempt to train the GPN from nothing.

As a baseline, we use two random generation algorithms:

- 1) **Naive**: The tiles are selected randomly with some weights. The marginal player probability is $1/(\text{width} \times \text{height})$, the marginal crate probability (and similarly for goals) is $1/\max(\text{width}, \text{height})$. The goal probability is independent on 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**: A random integer C is sampled uniformly from the range $[1, \max(\text{width}, \text{height})]$. 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 remaining 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. This ensures that the generated level satisfies the compilability requirements except for the requirement that the level is not already solved.

For time comparisons, all the experiments were conducted on a machine equipped with an Intel Core i7-5820k @ 3.3GHz (6 Cores, 12 Threads) processor with an Nvidia GTX 980 Ti GPU.

VIII. RESULTS AND DISCUSSION

In this section, we will show the results and compare the methods in each of the following aspects: Quality, Diversity, Training time, Generation time and Controllability. Then we will check if VAEs and VAEGANs are potentially useful for mixed-initiative cooperative level design by testing for the levels’ change percentage and quality improvement during reconstruction. Finally, we will discuss some aspects of the methods that are not shown in the experimental results.

TABLE II
QUALITY AND DIVERSITY STATISTICS
(DS: DIVERSITY SAMPLING, AT: AUXILIARY TARGETS, GMM: SAMPLING CONDITIONS FROM THE GAUSSIAN MIXTURE MODEL)
(THE SYMBOL $\sim$ MEANS THAT THE GENERATION WAS DONE VIA A STOCHASTIC POLICY)

Name	DS	AT	GMM	Playable	Duplicates	Tile Diversity	Unique Solutions	Unique Sig.	Copies
Random	Naive			0.01%	0.0%	0.00	1.0	1.0	-
	Compilable			5.2%	0.0%	0.38	524.0	195.0	-
GAN	No Bootstrapping			21.5% $\pm$ 21.2%	74.8% $\pm$ 42.2%	0.11 $\pm$ 0.13	22.8 $\pm$ 18.2	7.0 $\pm$ 5.9	50.6% $\pm$ 47.3%
	.	.	.	28.0% $\pm$ 10.6%	39.0% $\pm$ 9.2%	0.35 $\pm$ 0.05	175.0 $\pm$ 93.3	70.2 $\pm$ 28.2	0.22% $\pm$ 0.30%
	.	.	✓	54.1% $\pm$ 11.7%	42.0% $\pm$ 7.6%	0.33 $\pm$ 0.04	182.4 $\pm$ 102.6	73.6 $\pm$ 38.4	0.20% $\pm$ 0.27%
	.	✓	.	73.8% $\pm$ 4.1%	18.2% $\pm$ 9.4%	0.40 $\pm$ 0.04	136.0 $\pm$ 45.0	41.8 $\pm$ 16.8	0.07% $\pm$ 0.07%
	.	✓	✓	81.3% $\pm$ 3.8%	17.9% $\pm$ 8.9%	0.37 $\pm$ 0.03	124.0 $\pm$ 41.7	36.2 $\pm$ 17.9	0.11% $\pm$ 0.10%
	✓	.	.	29.7% $\pm$ 8.2%	67.0% $\pm$ 7.1%	0.29 $\pm$ 0.08	234.6 $\pm$ 39.4	151.2 $\pm$ 32.1	0.01% $\pm$ 0.02%
	✓	.	✓	37.7% $\pm$ 9.8%	64.1% $\pm$ 10.5%	0.29 $\pm$ 0.06	244.8 $\pm$ 43.4	154.8 $\pm$ 36.5	0.02% $\pm$ 0.05%
	✓	✓	.	57.0% $\pm$ 12.1%	37.9% $\pm$ 5.1%	0.36 $\pm$ 0.07	665.8 $\pm$ 184.3	456.6 $\pm$ 124.7	0.00% $\pm$ 0.00%
VAEGAN	✓	✓	✓	61.1% $\pm$ 12.3%	31.8% $\pm$ 9.8%	0.36 $\pm$ 0.08	641.0 $\pm$ 161.5	424.2 $\pm$ 111.5	0.00% $\pm$ 0.00%
	No Bootstrapping			38.0% $\pm$ 5.5%	86.2% $\pm$ 2.8%	0.32 $\pm$ 0.03	134.0 $\pm$ 45.7	32.6 $\pm$ 12.2	67.7% $\pm$ 9.9%
	.	.	.	53.1% $\pm$ 8.2%	32.9% $\pm$ 8.7%	0.30 $\pm$ 0.04	149.6 $\pm$ 88.4	46.0 $\pm$ 33.2	0.24% $\pm$ 0.19%
	.	.	✓	79.0% $\pm$ 5.4%	37.8% $\pm$ 9.3%	0.29 $\pm$ 0.04	135.6 $\pm$ 79.9	42.8 $\pm$ 26.5	0.32% $\pm$ 0.26%
	.	✓	.	75.1% $\pm$ 12.6%	27.6% $\pm$ 8.4%	0.33 $\pm$ 0.04	134.2 $\pm$ 61.6	37.2 $\pm$ 29.9	0.01% $\pm$ 0.02%
	.	✓	✓	87.6% $\pm$ 4.8%	23.5% $\pm$ 4.2%	0.32 $\pm$ 0.02	100.2 $\pm$ 38.4	23.8 $\pm$ 13.2	0.04% $\pm$ 0.06%
	✓	.	.	49.6% $\pm$ 10.6%	50.7% $\pm$ 16.3%	0.32 $\pm$ 0.02	251.0 $\pm$ 71.6	126.4 $\pm$ 21.1	0.14% $\pm$ 0.08%
	✓	.	✓	57.7% $\pm$ 9.7%	53.0% $\pm$ 18.1%	0.32 $\pm$ 0.02	237.0 $\pm$ 76.9	117.4 $\pm$ 27.2	0.17% $\pm$ 0.12%
VAE	✓	✓	.	69.5% $\pm$ 7.8%	51.8% $\pm$ 16.6%	0.30 $\pm$ 0.01	331.6 $\pm$ 200.2	151.0 $\pm$ 96.6	0.00% $\pm$ 0.01%
	✓	✓	✓	79.1% $\pm$ 7.4%	48.5% $\pm$ 13.0%	0.30 $\pm$ 0.02	348.6 $\pm$ 163.0	164.2 $\pm$ 86.6	0.00% $\pm$ 0.01%
	No Bootstrapping			34.3% $\pm$ 4.0%	72.5% $\pm$ 4.0%	0.37 $\pm$ 0.01	406.0 $\pm$ 38.9	49.8 $\pm$ 8.3	70.1% $\pm$ 4.6%
	.	.	.	30.4% $\pm$ 6.7%	28.0% $\pm$ 4.2%	0.30 $\pm$ 0.02	492.6 $\pm$ 130.1	171.0 $\pm$ 70.4	0.22% $\pm$ 0.11%
PCGRL	.	.	✓	52.5% $\pm$ 5.0%	29.4% $\pm$ 5.7%	0.28 $\pm$ 0.02	519.0 $\pm$ 134.7	142.2 $\pm$ 53.0	0.25% $\pm$ 0.03%
	✓	.	.	32.4% $\pm$ 5.2%	39.1% $\pm$ 7.5%	0.31 $\pm$ 0.01	623.4 $\pm$ 123.7	262.0 $\pm$ 100.6	0.17% $\pm$ 0.08%
	✓	.	✓	46.4% $\pm$ 0.8%	37.6% $\pm$ 6.5%	0.29 $\pm$ 0.01	675.6 $\pm$ 140.3	256.0 $\pm$ 105.8	0.26% $\pm$ 0.12%
	Narrow			58.4%	0.0%	0.42	5758.0	1869.0	-
GPN	Narrow $\sim$			65.8%	0.0%	0.40	6527.0	2552.0	-
	Turtle			63.2%	0.0%	0.50	6316.0	2513.0	-
	Turtle $\sim$			85.9%	0.0%	0.43	8592.0	4287.0	-
	Wide			0.1%	0.0%	0.52	5.0	4.0	-
	Wide $\sim$			79.1%	4.9%	0.18	6234.0	1792.0	-
	C. Turtle			68.3%	0.1%	0.54	6769.0	216.0	-
	C. Turtle $\sim$			86.2%	3.7%	0.49	7955.0	162.0	-
				48.7%	99.4%	0.08	5.0	2.0	36.93%

A. Quality and Diversity

To evaluate the quality and diversity of each generator, we generated 10,000 levels from each one. For each run of the generative models, the results are computed separately then the mean and the standard deviation is reported. Copies are the percentage of playable levels that are exactly the same as a level found in the initial dataset. For GPN, the initial dataset is only the 12 levels shown in Fig.2 while the other generative models are compared to expanded dataset (167 levels). Tile diversity is the average hamming distance between all pairs of playable levels in the sample.

TABLE II shows the statistics related to the quality and diversity of each generator. We use DS, AT and GMM to denote Diversity Sampling, Auxiliary Targets and Gaussian Mixture Models respectively. The first two rows shows the results of the random baselines. For the naive random method, only a single level (out of 10,000) was playable. For the compilable random method, 524 levels were playable with 524 different solutions and 195 unique signatures. Although the playability rate is low, it was able to create enough unique levels that rivals the diversity of more than half the generative models in our experiments.

For GAN, VAE and VAEGAN, the first row in their sections of the table includes the results of running without bootstrapping, diversity sampling, auxiliary targets or sampling conditions from a Gaussian mixture model. For GAN and VAEGAN, training without bootstrapping decreases both the quality and the diversity since the percentage of playable levels decreases and so does the number of unique solutions and signatures. In addition, the amount of duplicated levels and levels copied from the dataset are high. For VAE, however, the playability percentage is higher than with bootstrapping (ignoring when the GMM was used). The decrease in the number of solutions is not huge but it is more significant for the signatures. Still, the number of duplicates and copies from the dataset is high even though the tile-wise diversity increased.

The results shows the effect of diversity sampling, auxiliary targets and GMM sampling. We can notice the following:

- In all cases, diversity sampling increases the number of unique solutions and signatures. It also decreases the percentage of levels copied from the initial dataset. However, it increases the duplication rate and in most cases, decrease the playability rate.

Fig. 5. Samples to Demonstrate the Effect of Diversity Sampling

- For auxiliary targets, it increases the playability rate in all cases. It also decreases the duplication rate and when used with diversity sampling, it increases the number of unique solutions and signatures. But without diversity sampling, auxiliary targets decrease the number of unique solutions and signatures. It is also noteworthy that in most cases, it decreases the coefficient of variation of the playability (the standard deviation divided by the mean) which possibly means that the training stability improves with the auxiliary targets.
- Sampling from conditions from a GMM also increases the playability rate in all cases. Its effect on the other metrics is neither consistently positive nor negative.

When comparing between GANs, VAEGANs and VAEs, VAEGAN has the highest playability rate and even without auxiliary targets and GMMs, it exceeds 50%. However, Both GANs and VAEs outperform VAEGANs in some diversity metrics such as the number of unique solutions and signatures; Even without diversity sampling, VAEs exceed all the VAEGAN configurations. It is noteworthy that without diversity sampling, VAEs outperform GANs in generating unique solutions and signatures. And with diversity sampling, VAEs still outperform GANs in both quality and diversity when no auxiliary targets are used. However, adding both diversity sampling and auxiliary targets present a significant gain for GANs at which point it outperforms the signature diversity of VAEs (while becoming on par regarding the solution diversity).

Fig.5 shows a random sample (after removing duplicates) of the levels generated by GANs (with auxiliary targets) with and without diversity sampling. As reported in TABLE II, Diversity sampling had the most prominent effect on GANs with auxiliary targets. As seen in Fig.5, both samples have a high variety in the tiles, however, the solutions are significantly different when diversity sampling is applied. It

is also noteworthy that only a few of the tile changes have a significant effect on the solution. Another note is that many useless crates were added by the generator which seems to be how the generator cheats its way to satisfy the crates condition without risking to break the level.

For uncontrollable PCGRL, almost all three agents generate high quality levels with a significantly higher number of unique solutions and signatures compared to the generative models. The only exception is the wide representation agent, when run as a deterministic policy, where it gets stuck in a loop after 2 or 3 steps. Even for the other representations, there is a significant drop in quality when the agent acts deterministically. For both the turtle and the narrow agent, every level is unique and almost all of them have unique solutions. On the other hand, the wide representation has a few duplicates but much less than any generative model. Among all the methods, the turtle representation exhibits superior playability (insignificantly less than the best VAEGAN) while producing the highest number of unique solutions and signatures (more than 9 times the unique signatures produced by any generative model).

For controllable PCGRL, it achieved the highest playability percentage among the PCGRL agents and it is superior to almost all the generative models (insignificantly less than the best VAEGAN). It also has the highest tile diversity despite having a few duplicates (compared to zero duplicates from the other agents). Most of the levels also have unique solutions, however, the number of unique signatures is quite low in comparison. Upon inspecting a random sample of the generated levels as shown in Fig.6, we noticed that the agent learned to add crates and goals in adjacent pairs where every crate most likely has a goal on the tile above it. Therefore, most of the solutions are expected to have signatures that are just repeated strings of the action "U" (push crate up). It is probable that this issue will be fixed if we limit the change

Fig. 6. Samples of Solvable Levels from Controllable PCGRL

Fig. 7. Samples of Solvable Levels from GPN

percentage during training similar to the other PCGRL agents.

For GPN, the playability rate is relatively high compared to some other generative models, however, most of the levels are duplicates and the tile diversity, unique solutions and unique signatures are very low. Upon inspecting a sample of generated levels as shown in Fig.7, we noticed that most of the levels are minor modifications of two levels seen in the original dataset used for the agent pretraining. In an attempt to explain the results, we tested the agent on all 12 levels of the dataset. The agent only won 5 out of the 12 levels. These 5 levels were trivial but only two levels required significantly more steps compared to the others. Thus these levels became the basis for all the generated levels during the training process and the agent failed to learn how to win any levels with more complexity. Thus, we believe the bottleneck for the GPN is

the agent since it is hard for reinforcement learning agents to learn Sokoban solving. It is noteworthy that we tested other architectures for the GPN agent including the Deep Repeated Convolution-LSTM [29] which was able to learn the game much faster but failed to supply useful gradients for the generator. Based on these results, we did not attempt to train GPN from nothing since we have low expectations given the current performance of the agent. To save space, the GPN will be opted out from most of the results for the rest of this section.

Overall, VAEGAN presents the best option regarding the playability rate but at a huge sacrifice of diversity. In our opinion, the GAN model with diversity sampling and auxiliary targets presents the best compromise out of the generative models and also has an advantage over controllable PCGRL

(a) Solutions without Diversity Sampling

(b) Solutions with Diversity Sampling

Fig. 8. Re-occurrence of Solutions in different outputs

(a) Signatures without Diversity Sampling

(b) Signatures with Diversity Sampling

Fig. 9. Re-occurrence of Signatures in different outputs

in both playability and signature diversity. However, the uncontrollable PCGRL models significantly outperform the generative models diversity-wise while being on par with them in playability rate. Thus, when control is not required, PCGRL presents the best option regarding quality and diversity.

Since we trained every generative model 5 times on the same dataset, we were curious to check the overlap between the generated levels from the 5 trained models. Fig.8 and Fig.9 shows how many solutions and signatures are common among multiple models. If a solution or a signature appear in the output of N models only, it is added the N th pin in the histogram. We created the same histogram for the levels but, to our surprise, almost 100% of the generated levels were unique to their model and the percentage of common levels never exceeded 6%. Therefore, we decided not to include their histograms to save space. This note does not apply to the models trained without bootstrapping where the percentage of common levels were 42%, 60% and 78% for GANs, VAEGANs and VAEs respectively. By inspecting Fig.8 and Fig.9, we notice that the diversity sampling increases the number of solutions and signatures that are unique for each run. Also the number of signatures that are common among all 5 runs decreases, but surprisingly, the number of solutions that appear in the 5 runs have different changes (some increase while others decrease) across different models. Thus for solutions, most of them either appear only once or appears in all 5 runs when diversity sampling is employed.

To further explore the generators' diversity, we plot the expressive range of each generator (without GMM) in the 2D behavior space of the number of crates and the solution length. For generative models, we plot the range of all the 5 runs in a single plot. For most of them, the expressive range has minor changes across the runs, however, we will later show some examples where that is not the case. Fig.3 shows the initial dataset distribution in the behavior space where the mode is at $[crates : 1, sol - length : 1 - 10]$ and the crates range from 1 to 3 and solution length range from 1 - 10 to 31 - 40 (precisely, 1 to 31). By inspecting Fig.11, Fig.12 and Fig.13, we notice

Fig. 10. Expressive Range of Compilable Random

that all the generative models has an expressive range that exceed the initial dataset and in some cases, the mode shifts to a different bin.

Out of all models, GANs with auxiliary targets and diversity sampling has the widest expressive range with some levels exceeding 10 crates and others exceeding 100 steps for the shortest solution. It is noteworthy that diversity sampling shifts the mode from the bottom-left bin to around 31-40 on the solution length axis and to around 3 to 4 on the crates axis. Fig.11b shows that diversity sampling have a negative effect on the expressive range over the number of crates. It is not unexpected since it is not necessary to add many crates to find levels with a big variety of solutions. This is even more cemented by the fact that the uncontrollable PCGRL agents in Fig.14 do not cover a large range on the crates axis.

In Fig.12 and Fig.13, a similar effect of diversity sampling is observable where the mode shifts out towards higher solution lengths and crates. It is also observable that the effect of diversity sampling is less prominent on VAEs compared to

(a) GAN

(b) GAN DS

(c) GAN AT

(d) GAN AT DS

Fig. 11. Expressive Range of GAN

(a) VAE

(b) VAE DS

Fig. 12. Expressive Range of VAE

(a) VAEGAN

(b) VAEGAN DS

(c) VAEGAN AT

(d) VAEGAN AT DS

Fig. 13. Expressive Range of VAEGAN

GANs. In Fig.13d, it is notable that diversity sampling has a similar effect on VAEGAN (with auxiliary target) as it has on GANs in Fig.11b where the range on the crates axis decreased accompanied by more expansion on the solution length axis. In both Fig.11 and Fig.13, model trained with auxiliary targets exhibits more expansion on the crates axis. A possible explanation is that auxiliary targets improves the model's ability to learn the equality between the number of crates and goals.

For uncontrollable PCGRL, all three representations have mostly similar ranges as shown in Fig.14 but the levels are more distributed the narrow representation as shown in Fig.14a. Their ranges barely extends to more than 5 crates and the main reason is that the maximum crates were set to 5 during training so the agents prefer not to go beyond 5 crates. In Fig.14d, the expressive range of the controllable turtle agent shows more expansion on the crates axis up to 10 crates but it fails to generate levels with long solutions using a few crates. Finally, the range of the compilable random method in Fig.10 is limited on the crates axis from 1 to 7 which is enforced by its sampling process and it is limited to solutions no longer

than 60 steps which is lower than all the other methods except VAEGAN with auxiliary targets where the maximum solution length was 52.

Fig.15 shows an example of how sampling conditions from a GMM affect the expressive range. While GMMs increase the chance of sampling a playable level, they could slightly limit the expressive range of the generator. As seen in Fig.15, some of the bins at the extremes of the expressive range are no longer occupied with any samples since the GMM assigns a low probability for sampling conditions from these bins. Fig.15c shows the probability of sampling a condition from each bin when the GMM is used.

Fig.16 shows examples of the expressive range variations across different runs of the generative models. For GANs without diversity sampling and auxiliary targets, we can notice that only one of the 5 experiments had an expressive range that covers crates exceeding 10 crates. For the other 4, they have a much tighter range with different shapes across runs. With diversity sampling and auxiliary targets, GANs have a higher chance of expanding to wide range where only 1 out of 5 runs had a tight range. For VAEs, the shape and range

(a) PCGRL Narrow

(b) PCGRL Turtle

(c) PCGRL Wide

(d) PCGRL Controllable Turtle

Fig. 14. Expressive Range of PCGRL (Stochastic)

(a) Expressive Range Without GMM

(b) Expressive Range With GMM

(c) Marginal GMM

Fig. 15. Effect of GMM Sampling on The Expressive Range of GAN with Diversity Sampling and Auxiliary Targets

Fig. 16. Variations in Expressive Range across Runs

of the generator is almost the same across runs. It seems that VAEs have the least variation across runs and the same was noticed even without diversity sampling. It is also worth noting that in Fig.8 and Fig.9, VAEs was, in most cases, the model where the most solutions and signatures appearing across the 5 runs. VAEGANs also exhibit notable variations across runs. So overall, it seems that VAEs have the least amount of variations across runs which can also be noticed by looking at the standard deviations in TABLE II.

B. Training and Generation Time

TABLE III shows the training and generation time for each generator. It is quite apparent that the bootstrapping method is much faster to train. The longest training time was 37 minutes and 33 seconds which belongs to a GAN with diversity sampling and auxiliary targets. Most of the other runs finish in around 20 minutes and VAE takes even less time. This is considerably a great benefit for the bootstrapping method compared to the other methods which requires training for days on our machine. For PCGRL, the training time mainly depends on the number of changes issued by the agent since the solver does not need to run in steps where no changes occur. The narrow and wide representation required more training time since the action space consists of edit actions only. For the turtle representation, the agent quickly learns to move around and therefore most of the steps do not include any changes. The narrow and the wide representation takes more time to learn that behavior. The controllable PCGRL agent is trained for $5\times$ the steps used to train its uncontrollable counterpart. However, we used the optimized solver so the training time was only around twice the training time of the

uncontrollable turtle agent. For GPN, it took 2 days and 18 hours mainly due to the larger network architectures and the slow communication with the GVGAI environments.

TABLE III also includes the inference times for a single network evaluation call. The time is calculated by supplying a random input (batch size = 1) for 10,000 iterations and recording the average run time. All of the generative models have the same time since they all use the same generator architecture. For PCGRL, only the network for the wide representation is different enough to have a notable impact on the inference time. Both GPN and the generative models require a single forward call to create a level. For the PCGRL agents, we recorded the number of steps used to generate each level. While acting deterministically, the turtle agent require less steps compared to the narrow agent since it can control its movement and thus reach its target tile faster. Also, the wide representation finishes quickly since it gets stuck after a couple steps at which point it is terminated. While acting stochastically, the episode takes much longer for all representations since we no longer terminate the episode when a state is revisited. It is noteworthy that the step limit is $49^2 = 2401$ steps. The narrow representation takes less time since the action space only contains tile editing actions so it is more likely to hit the change limit early on. The controllable turtle also has a high chance of finishing early since it terminates as soon as the targets are achieved.

In general, the generative models are the fastest to train. For generating a level, the generative models and GPNs are the fastest since they require a single forward propagation call only to create a level.

TABLE III
TRAINING AND GENERATION TIME

Name	DS	AT	Training	Inference	Inferences / Level	Level Generation
GAN	.	.	15 : 48s ± 06 : 56s	2.08ms	1	2.08ms
	.	✓	9 : 40s ± 03 : 35s	2.08ms	1	2.08ms
	✓	.	20 : 28s ± 09 : 24s	2.08ms	1	2.08ms
	✓	✓	21 : 41s ± 11 : 36s	2.08ms	1	2.08ms
VAEGAN	.	.	18 : 37s ± 00 : 16s	2.08ms	1	2.08ms
	.	✓	20 : 32s ± 00 : 17s	2.08ms	1	2.08ms
	✓	.	18 : 40s ± 00 : 09s	2.08ms	1	2.08ms
	✓	✓	20 : 30s ± 00 : 23s	2.08ms	1	2.08ms
VAE	.	.	7 : 14s ± 01 : 16s	2.08ms	1	2.08ms
	✓	.	8 : 10s ± 02 : 42s	2.08ms	1	2.08ms
PCGRL	Narrow					
	Narrow (Stochastic)		4 days 00 : 15 : 19s	0.82ms	127.0 ± 28.3 1110.1 ± 315.2	104.1ms ± 23.2ms 910.3ms ± 258.5ms
	Turtle					
	Turtle (Stochastic)		2 days 08 : 02 : 17s	0.83ms	54.8 ± 15.8 2381.7 ± 203.8	45.5ms ± 13.1ms 1976.8ms ± 169.2ms
	Wide					
GPN	Wide (Stochastic)		3 days 01 : 18 : 56s	1.28ms	2.4 ± 0.7 2397.4 ± 58.8	3.1ms ± 0.9ms 3068.7ms ± 75.3ms
	C. Turtle					
	C. Turtle (Stochastic)		4 days 18 : 39 : 05s	0.83ms	70.2 ± 27.8 1013.9 ± 963.7	58.3ms ± 23.1ms 840.8ms ± 799.9ms
GPN			2 days 18 : 19 : 48s	1.76ms	1	1.76ms

C. Controllability

To measure the control error of each generator, we request 100 levels for every possible control target in the grid: $crates \in [1, 10]$, $solution\ length \in [1, 100]$. The other conditions (player distance from center and wall ratio) were sampled from GMM conditionally given the fixed conditions. It is notable that the playability rates in TABLE IV is different from those in TABLE II since we sample conditions from outside of the final training dataset range in this experiment. The error is the absolute difference between the target and the actual properties of the generated level. The crates error is calculated for all the generated levels and the solution length error is calculated for the playable levels only. Therefore, we also report the percentage of playable levels since they seem to affect the mean solution length error.

TABLE IV shows the control error for each of the controllable models in our experiments which include the generative models and the controllable PCGRL agent. It is notable that the increase in the playability percentage usually leads to an increase in the solution-length error. For the models with relatively high playability, GANs with diversity sampling and auxiliary targets have the lowest solution length error. In comparison, PCGRL presents a significant improvement in the playability rate but at a notable increase in the solution-length error. It seems that using auxiliary targets without diversity sampling and vice versa causes a degradation in the solution length error. And for PCGRL, acting stochastically improves both the control error and the playability.

To better understand the controllability of each model, we visualize the confusion matrices for the crates and the solution lengths. Fig.17 shows the confusion matrices of the crate control and Fig.18 shows the confusion matrices of the solution length control. We use DS and AT to denote Diversity Sampling and Auxiliary Targets respectively. The matrices

TABLE IV
CONTROL ERROR
(THE SYMBOL $\sim$ MEANS THAT THE GENERATION WAS DONE VIA A STOCHASTIC POLICY)

Name	DS	AT	Error		
			Crate	Sol-Length	Playable
GAN	.	.	2.8 ± 1.3	5.8 ± 3.2	21.7% ± 20.5%
	.	✓	2.2 ± 0.7	16.7 ± 18.7	45.2% ± 26.5%
	✓	.	2.9 ± 0.6	11.6 ± 3.8	25.8% ± 13.3%
	✓	✓	2.3 ± 0.4	10.1 ± 3.0	48.1% ± 19.3%
VAEGAN	.	.	3.5 ± 0.6	4.8 ± 1.2	20.3% ± 5.5%
	.	✓	7.3 ± 7.0	21.4 ± 17.9	39.3% ± 29.9%
	✓	.	3.0 ± 0.4	8.7 ± 4.6	28.7% ± 5.6%
	✓	✓	3.1 ± 0.3	10.5 ± 4.5	49.3% ± 11.6%
VAE	.	.	3.1 ± 0.5	6.2 ± 2.0	23.7% ± 8.0%
	✓	.	2.6 ± 0.3	7.1 ± 1.3	24.8% ± 4.8%
PCGRL	C. Turtle		2.6	21.9	68.9%
	C. Turtle $\sim$		2.5	17.5	86.6%

include all the generated levels, even unplayable ones. For the solution length confusion matrices, the first column is dedicated for unplayable levels.

In Fig.17, we notice that for GANs, the matrix is more diagonal-like when auxiliary targets are used. On the other hand, Using diversity sampling adds some confusion to the matrix and the GANs' confusion matrix is at its most distorted form when diversity sampling is used alone. For VAEs, we also notice a diagonal-like shape and in this case, it seems to slightly improve when diversity sampling is used. On the other hand, VAEGANs seems to be unable to handle the crate control. The PCGRL agent seems to have some control over the crates, however, the highest probability in all the rows lie around the same range (3–5 while acting deterministically, 5–7 while acting stochastically). In general, GANs with auxiliary

Fig. 17. Crates Confusion Matrices

targets seems to have the least confusion, but overall, all the matrices are pretty fuzzy and the models frequently fail to satisfy the crate target.

For the solution length, it seems that most models are performing well for most target values but they tend to fail if the requested solution length is high as seen in Fig.18. In such case, they either create levels with lower solution lengths (such as the PCGRL agent) or they create unsolvable levels (such as plain VAEGAN). It is noteworthy that training with auxiliary targets without diversity sampling leads to the least performance regarding the solution-length control. Based on Fig.18, it seems that GANs with diversity sampling and auxiliary targets has the least confusion over the solution-length target if we ignore the unplayable levels.

Overall, it seems that most models are more capable at controlling the solution length compared to controlling the number of crates which is surprising given that it seems intuitive that satisfying the crates target should be easier than controlling the solution length. For PCGRL, it is possible that after the agent becomes better at creating solvable levels, the rewards for reaching the solution-length target overwhelms the crates target reward. So it could be possible that the agent learned to put less focus on the crates target and focus more on the solution-length target. Anyway, based on TABLE IV, Fig.17 and Fig.18, it seems that GANs with diversity sampling

and auxiliary targets have the best control over their output throughout the control domain.

D. Level Reconstruction and Repair

TABLE V shows the results of using VAEGAN and VAE for level reconstruction and repair. The idea is inspired by another work [30] where autoencoders are used to generate, repair and recognize Mario levels. Initially, we wanted to test if the models can take levels that they have never seen before and reconstruct it with minimal changes and hopefully fix that level if it was unplayable. The first 4 columns show the results when the input levels are sampled from the output of the PCGRL turtle agent which should be out-of-distribution for the tested models. For each playable level, it is supplied to the model with its actual condition. For each unplayable level, we test the level 10 times where in each test, the condition is randomly sampled from the GMM since conditions cannot be extracted from unplayable levels. The last 4 columns shows the results when the levels are sampled from the output of the model under test and thus we describe these data as in-distribution. The testing process is the same as described with the out-of-distribution data. From out-of-distribution data, we randomly sample 200 levels (100 unplayable and 100 playable levels) and use them for all the tests. We sample the same number of

Fig. 18. Solution Length Confusion Matrices

TABLE V
RECONSTRUCTION

Name	Input State:		Out-of-Distribution				In-Distribution			
			Unplayable		Playable		Unplayable		Playable	
	DS	AT	Similarity	Fixed	Similarity	Unbroken	Similarity	Fixed	Similarity	Unbroken
VAEGAN	.	.	40.3%	85.2%	43.4%	42.6%	67.0%	84.6%	91.9%	89.8%
	.	✓	40.9%	96.6%	44.0%	60.0%	65.0%	95.9%	88.7%	98.0%
	✓	.	39.0%	62.0%	45.8%	57.2%	67.7%	62.1%	90.7%	69.6%
	✓	✓	42.8%	89.9%	46.0%	79.4%	66.7%	89.5%	87.6%	92.8%
VAE	.	.	44.0%	50.6%	49.6%	41.0%	70.7%	53.5%	98.7%	88.2%
	✓	.	42.8%	45.8%	48.5%	51.4%	68.5%	46.7%	98.4%	86.8%

levels for the in-distribution data tests. Similarity is measured as the percentage of unchanged tiles.

The first two columns in TABLE V represents the most common use case where the level design process is still in progress so the level would most likely be both unplayable and never seen in the model’s output (out-of-distribution). Anyway, regardless of whether the level was playable or not, the similarity rate is too low for all the out-of-distribution levels as the model changes more than 50% of the tiles. For in-distribution data, the similarity is relatively higher and less tiles are changed if the level was already playable. Also, there is a low chance (in most cases) that a playable in-distribution level will be broken during reconstruction. Overall,

VAE without diversity sampling has the highest similarity rates while the VAEGAN models with auxiliary targets have the highest chances of returning a playable level. Overall, the results shows that our VAEs and VAEGANs are not suited to be repair tools for mixed-initiative level design since they will tend to overwrite most of the user’s design choices.

In comparison, PCGRL agents are inherently fit for mixed-initiative level design since they are trained to achieve playable levels using a limited number of changes. In addition, it is possible to set an upper limit on the number of changes during generation. RL Brush [18] employed PCGRL agents for mixed-initiative level design and showed that users, who used the AI suggestions, created more playable and more complex

Sokoban levels.

E. Discussion

There are some aspects of the generators that we did not experimentally test or report. First of all, only the bootstrapping method requires a dataset. However, it was shown that, with the right configuration, the model will expand the expressive range far beyond the initial dataset. So maybe, we can use one of the two random generation methods and randomly generate the initial dataset. Even the naive random generator can create 10 levels in less than a million trials within a couple of minutes.

Another thing we would like to note is that GANs are very sensitive to any changes in the hyperparameters. This is time consuming since it takes a significant number of experiments to fine-tune the network architecture, learning rate, etc. In comparison, it is easy to design and pick hyperparameters that work well for VAEs. This is a huge selling point for VAEs especially when the game specifications are under development (subject to frequent updates) and the generator is being used to explore the level space.

IX. CONCLUSION

Procedural level generation via deep learning has seen a lot of progress in the recent years. However, there has been only a few comparisons between the different techniques for various reasons. In this paper, we picked level generation for Sokoban at size 7×7 as a benchmark and picked four of the recent methods for an experimental study. For bootstrapping conditional GANs [9], we adapted their framework for Sokoban then we added VAEs and VAEGANs to the list of models to test. To increase the number of unique solutions and create levels with distinct plans, we proposed diversity sampling where levels are clustered using a distilled version of the solution. To increase the quality of the generator's output, we proposed using auxiliary targets to improve the training process and using GMMs to sample conditions during generation. We also showed how GMMs can be used to sample conditions under partial control. For PCGRL, we discussed how we modified the training process to decrease the training time. For GPN, we discussed how the reward function can be modified to support puzzle games where the player cannot die.

The results showed that VAEGAN generates a high percentage of playable levels but at the cost of diversity. On the other hand, GANs presented a good compromise between quality and diversity when both diversity sampling and auxiliary targets were used. We also showed that diversity sampling consistently increases the numbers of generated solutions and signatures while auxiliary targets and GMM condition sampling consistently improved the quality of the generators' outputs. The results also showed that uncontrollable PCGRL achieves superior quality and diversity among all the models with the main drawback being the long training time. For controllable PCGRL, the results showed that it achieves high quality, high tile-wise diversity and a large number of unique solutions, however, most of the levels follow the same idea

since almost every crate was accompanied by a goal in an adjacent tile.

Then we discussed the controllability of each model and showed that GANs with diversity sampling and auxiliary targets exhibit good control over their output throughout the control domain. Finally, we showed that our VAEs and VAEGANs tend to change most of the tiles if the input level is not from the generator's output distribution. Thus they are not suitable for use as level repair tools in a mixed-initiative level design setting.

X. FUTURE WORK

In this paper, we created an experimental setup to test some of the most recent deep-learning level generation methods but there are some techniques that we did not include such as LSTMs [16] and Adversarial Reinforcement Learning [31]. We are interested in adapting these methods to our experimental setting. In addition, we are interested in including more games from different genres to our upcoming experiments.

Out of the methods we tested, GPNs failed to achieve good results. We expect the bottleneck to be the weak agent performance so it is worth searching for a reinforcement learning agent that is capable of long term planning and can be used inside the GPN framework.

In the results, we showed that diversity sampling can improve the quantity of unique solutions in the generator's output, expand the expressive range and also increase the number of unique solution signatures. We argue that the signature is a useful representation to denote the level's unique idea from a player's perspective. However, this representation was designed based on our assumptions of how humans solve Sokoban levels. In addition, it is specific to Sokoban and not suited for other games. It would be extremely useful to find a generic representation of a level's idea and to conduct a user study to confirm whether such a representation is effective at modeling the perceptual uniqueness of levels or not.

The results of using VAEs and VAEGANs for level reconstruction and repair proved to be unsuited for mixed-initiative level design. More work is needed to expand their output distribution. In addition, it would be interesting to see if they can be used for AI-constrained co-creativity similar to how VAEs were used in Lode Encoder [32].

Compared to uncontrollable PCGRL, the generative models are lacking in quality and diversity but they are very fast to train and use. It would be useful to find methods to bridge the gap between these two approaches and combine their benefits.

REFERENCES

- [1] S. Risi and J. Togelius, "Increasing generality in machine learning through procedural content generation," *Nature Machine Intelligence*, pp. 1–9, 2019.
- [2] K. Compton. (2016, Feb.) So you want to build a generator.... [Online]. Available: <https://galaxykate0.tumblr.com/post/139774965871/so-you-want-to-build-a-generator>
- [3] A. Summerville, S. Snodgrass, M. J. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, pp. 257–270, 2018.

- [4] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2014. [Online]. Available: <http://arxiv.org/abs/1312.6114>
- [5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: <https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf>
- [6] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, pp. 95–101, Oct. 2020. [Online]. Available: <https://ojs.aaai.org/index.php/AIIDE/article/view/7416>
- [7] A. Summerville, "Expanding expressive range: Evaluation methodologies for procedural content generation," in *Proceedings of the 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2018*, 2018, pp. 116–122.
- [8] H. Imabayashi, "Sokoban," 1982.
- [9] R. Rodriguez Torrado, A. Khalifa, M. Cerny Green, N. Justesen, S. Risi, and J. Togelius, "Bootstrapping conditional gans for video game level generation," in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 41–48.
- [10] S. Earle, M. Edwards, A. Khalifa, P. Bontrager, and J. Togelius, "Learning Controllable Content Generators," in *2021 IEEE Conference on Games (CoG)*, 2021.
- [11] P. Bontrager and J. Togelius, "Learning to Generate Levels From Nothing," in *2021 IEEE Conference on Games (CoG)*, 2021.
- [12] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications*, vol. 33, no. 1, pp. 19–37, oct 2021. [Online]. Available: <https://link.springer.com/article/10.1007/s00521-020-05383-8>
- [13] B. De Kegel and M. Haahr, "Procedural puzzle generation: A survey," *IEEE Transactions on Games*, vol. 12, no. 1, pp. 21–40, 2020.
- [14] A. V. Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1747–1756. [Online]. Available: <https://proceedings.mlr.press/v48/oord16.html>
- [15] A. Sarkar, Z. Yang, and S. Cooper, "Conditional level generation and game blending," *ArXiv*, vol. abs/2010.07735, 2020.
- [16] M. Suleman, F. H. Syed, T. Q. Syed, S. Arfeen, S. I. Behlim, and B. Mirza, "Generation of sokoban stages using recurrent neural networks," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 3, 2017. [Online]. Available: <http://dx.doi.org/10.14569/IJACSA.2017.080364>
- [17] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, "Self-attention generative adversarial networks," in *36th International Conference on Machine Learning, ICML 2019*, 2019.
- [18] O. Delarosa, H. Dong, M. Ruan, A. Khalifa, and J. Togelius, "Mixed-initiative level design with rl brush," in *EvoMUSART*, 2021.
- [19] R. D. Gaina, A. Couëtoux, D. J. N. J. Soemers, M. H. M. Winands, T. Vodopivec, F. Kirchgeßner, J. Liu, S. M. Lucas, and D. Perez-Liebana, "The 2016 two-player gvgai competition," *IEEE Transactions on Games*, vol. 10, no. 2, pp. 209–220, 2018.
- [20] J. H. Lim and J. C. Ye, "Geometric gan," *ArXiv*, vol. abs/1705.02894, 2017.
- [21] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," *ArXiv*, vol. abs/1512.09300, 2016.
- [22] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier GANs," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 2642–2651. [Online]. Available: <https://proceedings.mlr.press/v70/odena17a.html>
- [23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 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 *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.
- [25] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSE: Neural Networks for Machine Learning, 2012.
- [26] S. Roberts, D. Husmeier, I. Rezek, and W. Penny, "Bayesian approaches to gaussian mixture modeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 11, pp. 1133–1142, 1998.
- [27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *ArXiv*, vol. abs/1707.06347, 2017.
- [28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1412.6980>
- [29] A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racanière, T. Weber, D. Raposo, A. Santoro, L. Orseau, T. Eccles, G. Wayne, D. Silver, and T. P. Lillicrap, "An investigation of model-free planning," *CoRR*, vol. abs/1901.03559, 2019. [Online]. Available: <http://arxiv.org/abs/1901.03559>
- [30] R. Jain, A. Isaksen, C. Holmgård, and J. Togelius, "Autoencoders for Level Generation, Repair, and Recognition," *ICCC Workshop on Computational Creativity and Games*, 2016.
- [31] L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, "Adversarial Reinforcement Learning for Procedural Content Generation," in *2021 IEEE Conference on Games (CoG)*, 2021.
- [32] D. Bhaumik, A. Khalifa, and J. Togelius, "Lode Encoder: AI-constrained co-creativity," in *2021 IEEE Conference on Games (CoG)*, 2021.

Yahia Zakaria Biography text here.

Mayada Hadhoud Biography text here.

Magda Fayek Biography text here.