

Path of Destruction: Learning an Iterative Level Generator Using a Small Dataset

Matthew Siper
Game Innovation Lab
New York University
New York, USA
ms12010@nyu.edu

Ahmed Khalifa
Institute of Digital Games
University of Malta
Msida, Malta
ahmed@akhalifa.com

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

Abstract—We propose a new procedural content generation method which learns iterative level generators from a dataset of existing levels. The Path of Destruction method, as we call it, views level generation as repair; levels are created by iteratively repairing from a random starting level. The first step is to generate an artificial dataset from the original set of levels by introducing many different sequences of mutations to existing levels. In the generated dataset, features are observations of destroyed levels and targets are the specific actions that repair the mutated tile in the middle of the observations. Using this dataset, a convolutional network is trained to map from observations to their respective appropriate repair actions. The trained network is then used to iteratively produce levels from random starting maps. We demonstrate this method by applying it to generate unique and playable tile-based levels for several 2D games (Zelda, Danger Dave, and Sokoban) and vary key hyperparameters.

Index Terms—procedural content generation, neural networks, supervised learning, games, level generation, data augmentation

I. INTRODUCTION

Procedural Content Generation (PCG) in games refers to algorithmic methods for generating game content such as quests, items, and levels. Such methods need to generate content that fulfill multiple criteria, both aesthetic and functional. In this paper, we address the question of how can we learn an *iterative* game level generator *from a dataset* of existing levels. Unpacking this statement requires to consider two recent developments in PCG research: PCG via machine learning (PCGML) and PCG via reinforcement learning (PCGRL).

PCGML refers to the use of machine learning, mainly through self-supervised learning on existing content, to learn content generators [1]. This provides an alternative to building content generators from scratch, either through ad-hoc methods or using search, optimization, or constraint satisfaction as a principle [2]–[4]. Content generators trained via self-supervised learning tend to produce a whole artifact (such as a level) at the same time; in particular, this is true for deep learning methods such as autoencoders [5] or generative adversarial networks [6]. In cases where a level has a natural sequential encoding, a sequence learning architecture such as n-grams [7]

or LSTM networks [8] can be used, but such encodings are not universally applicable.

PCGRL provides a different way of using machine learning to learn content generation [9], [10]. In the absence of training data, reinforcement learning is used to learn to generate by trial and error. The generator is seen as an agent that takes actions to change its environment, and at the end of an episode it gets rewarded based on the quality of the artifact created by its actions. Interestingly, the resulting generator is fundamentally different from those learned by self-supervised learning, as it produces content *iteratively*, action by action. Iterative content generators have different affordances than one-shot generators. In particular, it can be easier to build mixed-initiative systems around iterative generators [11]. However, reinforcement learning can be slow and unreliable, and designing appropriate rewards is a non-trivial task.

In cases where some prior content already exists, how could we learn an iterative content generator? This paper proposes one of several possible solutions to this problem and applies it to the generation of 2D game levels. Our approach is to artificially destroy levels, framing content generation as the problem of repairing these destroyed levels by learning a repair agent trained from the destruction process.

The first step in our approach is to destroy the levels, which is accomplished by iteratively making destructive edits on the goal map until it is transformed into a randomly generated starting map. This sequence of destructive edits forms the Path of Destruction (PoD), which is then used to form a dataset of desirable repair actions. In this dataset, each instance consists of a feature set, which is a part of the level *after* destruction, and a target, which is a part of the level *before* destruction. Using standard supervised learning methods, we can train a classifier to act as a repair agent by reversing the Path of Destruction. By judiciously choosing the feature set—i.e., the observation space of the agent—and other parameters, we can achieve a high degree of generalization, such that the trained agent not only repairs broken levels but also generates new playable levels from unseen randomized starting levels.

II. THE PATH OF DESTRUCTION

We introduce a technique which we refer to as the Path of Destruction (PoD) in order to train generators to produce unique and playable levels¹². As previously explained, PoD is a technique that consists of generating a training dataset by iteratively destroying goal levels until they reach random noisy levels. During each of these destructive sequences, a repair sequence is iteratively produced with each destructive step and added to the training trajectory, which is used to train the repair agent.

A. Training set generation

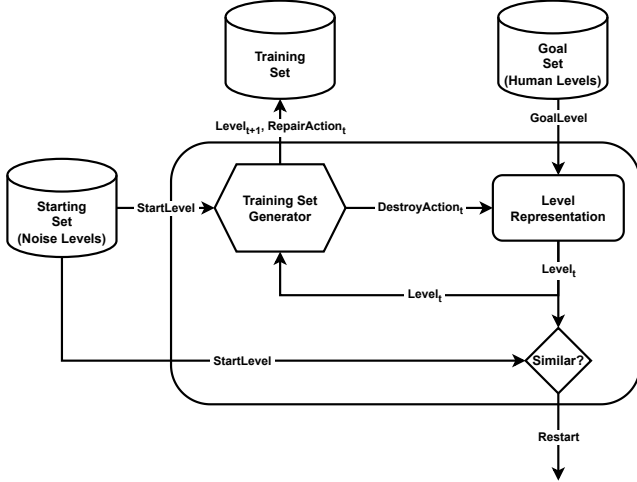


Fig. 1: System diagram explaining the training set generation step in the Path of Destruction method.

The steps for generating the training set is at the heart of the Path of Destruction technique. The goal of this step is to generate a large artificial dataset that can be used to train an agent to repair destroyed levels and create new levels by “repairing” from noise. Figure 1 shows the system diagram that illustrates the main loop of generating the training set. To start this step, we need two datasets:

- **Starting set:** This is a set of “levels” that will be used later as the starting level for generation. In this project, the starting set consist of noise (random tiles) that is produced from a predefined distribution such as a uniform distribution.
- **Goal set:** This is a set of levels that we want our generator to learn to repair towards them. In this project, the goal set is a set of human designed levels that are playable and unique.

From this dataset, we generate the training dataset by executing the following algorithm:

- 1) Generate a noisy random level from a predefined distribution ($StartLevel$).

- 2) Select the closest goal level ($GoalLevel$) from the goal set to the start level ($StartLevel$).
 - a) Calculate the hamming distance between each level in the goal set and the start level ($StartLevel$).
 - b) Select the level with the smallest hamming distance to be the goal level ($GoalLevel$)
- 3) Set the current level ($Level_t$) to be equal to the goal level ($GoalLevel$).
- 4) Destroy the current level ($Level_t$) until it is equal to the start level ($StartLevel$).
 - a) Select the next tile location on the level (can be random or sequential starting from location (0,0)).
 - b) Save the tile value at the selected location from both the current level ($RepairAction_t$) and from the start level ($DestroyAction_t$).
 - c) Update the current level ($Level_t$) with the tile value saved from the start level ($DestroyAction_t$).
 - d) Add the current level and the original tile value before destruction ($Level_t$, $RepairAction_t$) to the training set.
 - e) If the current level ($Level_t$) and the start level ($StartLevel$) are not the same then go back to step 4a.
- 5) Repeat the entire process until the training set reaches the appropriate size.

In our experiments, the destruction process (step 4) stops when the current level ($Level_t$) is the same as the start level ($StartLevel$). However, this is not a requirement. The only stopping requirement is that the current level ($Level_t$) is close to the starting set distribution. This allows the network to learn to repair (generate) levels from any sampled starting level that follows that distribution which is explained in section II-C. One could imagine the stopping criteria be when the current level ($Level_t$) is *close enough* to the start level ($StartLevel$) instead of identical or the tile distribution of the current level ($Level_t$) is close enough to that of the starting set.

B. Level generator training

For this project, we train a neural network that takes a cropped observation of the current level as input, and predicts the appropriate repair action. The action space is defined as the set of possible game tile values. The cropped observation consists of the tiles surrounding and including the current tile location. The observation is parameterized by a number, which we refer to as the crop size. The crop size determines the number of tiles surrounding the current tile location that are observable by the agent at each step. The input observation consists of the tiles cropped around the changed tile such that the changed tile is in the center (shown in figure 3). We experimented with different crop sizes as previous research [12], [13] showed that neural networks with local observation are cable of learning a more general strategy compared to networks that have access to the full observation.

C. Level generation

¹https://github.com/matt-quant-heads-io/path_of_destruction

²The name comes from the fact that the dataset is formed from a trace of destructive events (random tile changes), which is then retraced.

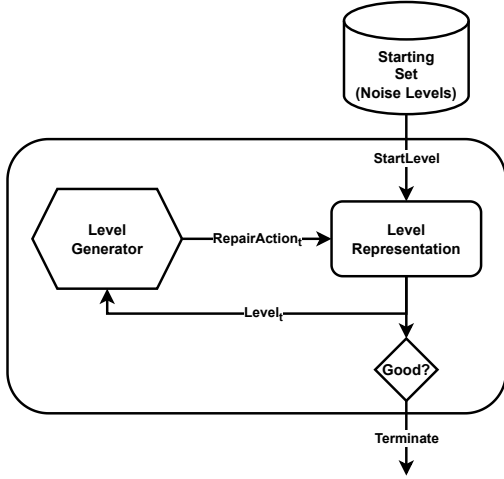


Fig. 2: System diagram explaining the level generation step in the Path of Destruction method.

The inference, or level generation step, is the final step of the Path of Destruction. This step focuses on converting a noisy starting level to a playable level. Figure 2 shows the main loop of generating a new level. The level generator modifies the current level iteratively with following these steps:

- 1) Initialize the current level ($Level_t$) via generating a noisy level from a predefined tile distribution.
- 2) Select the next tile location either randomly or sequentially starting from (0,0) location.
- 3) Feed the cropped current level ($Level_t$) to network, which outputs a repair action ($RepairAction_t$).
- 4) Update the current tile location in the current level ($Level_t$) with the repair action ($RepairAction_t$).
- 5) If the current level ($Level_t$) isn't playable and the number of steps hasn't reached the threshold, go to step 2.

This process enables the agent to transform a starting noisy level into a playable level ($Level_t$) over sequential iterations of repair. The stopping condition for the generation process is usually defined by the user based on the current application. In our project, we used playability as the stopping condition since we are trying to generate playable levels.

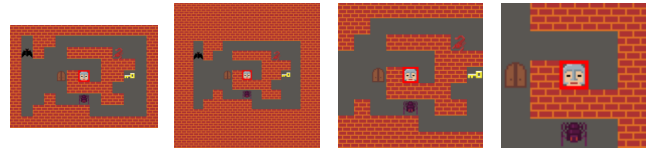
III. EXPERIMENTS

We conduct 4 different experiments. In the first experiment, we measure the effect of constraining the observation size. In the second experiment, we constrain the goal set size to measure its impact on agent performance. In the third experiment, we evaluate the robustness of the PoD technique by applying it to additional game types. Finally, we benchmark our PoD algorithm against a state of the art level generation method that uses Conditional Embedding Self-Attention Generative Adversarial Networks (CESAGAN) [14]. For our level generator, we used a convolutional neural network. The network consists of 3 convolution layers, each with 3×3 kernels of size 128, 128, and 256, respectively. A 2×2 max pooling layer is used after the second convolution layer, followed by a fully connected

layer and a softmax activation function. The network is trained using the Rmsprop algorithm with a batch size of 64 and a learning rate of 0.001. Training consists of 500 epochs and employs categorical cross entropy as the loss function. For every experiment, we trained 3 networks and calculated the mean and standard deviation to show training stability.

For the first two experiments, and the last experiment, we used the Zelda version from PCGRL as our test bed. The games tested and their playability conditions are described in section III-C. After we trained the level generators, we evaluated each agent's performance over 10,000 independent inference trials. We calculated the percentage of generated playable levels, as well as, the percentage of generated playable and unique levels. For measuring uniqueness, we used the hamming distance between all the playable levels (generated or goal) and removed all of the levels that were less than 10% different. For the benchmarking experiment, we computed the number of duplicate levels instead of similarity to align with the metrics used in the original CESGAN experiments.

A. Constraining observation size



(a) Actual Map (b) Obs Size 15 (c) Obs Size 9 (d) Obs Size 5

Fig. 3: Example of the different observation sizes for a certain goal map. The red rectangle is the current tile that is going to be acted upon by the agent.

In this experiment, we explored the effect of crop size in the training set on agent performance. As explained in section II, we converted the destroyed levels into observations by setting the center of the observation around the destroyed tile and cropping the surrounding tiles. This is similar to the narrow representation presented in the PCGRL framework [9]. To measure the effect of crop size on agent performance, we tested 3 different observation sizes: 15, 9, and 5. These sizes were selected based on the map size of Zelda (7×11). A crop size of 15 gave a full view of the level, whereas a crop size of 9 gave a partial view, and a crop size of 5 gave the most local view. Figure 3a shows the different observation sizes for the center location of the map. We hypothesized that a smaller observation size would help the trained agent to generate a diverse and unique set of playable levels.

B. Constraining goal set size

In this experiment, we sought to capture the effect of the size of the goal set used to generate the training data. We tested 3 different goal set sizes: 1, 5, and 50. We expected that a larger goal set size would help the agent produce a more diverse and unique set of playable levels. For each of these experiments, the observation size was fixed to 5.

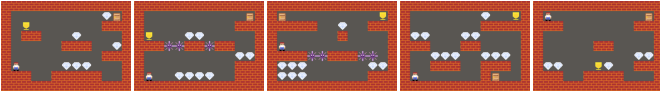
C. Different games



(a) Examples of Zelda levels from the goal set used to generate the training set.



(b) Examples of Sokoban levels from the goal set used to generate the training set.



(c) Examples of Danger Dave levels from the goal set used to generate the training set.

Fig. 4: Goal sets for different games used in training

In this experiment, we used the PoD method to train a level generator across 3 different games. Figure 4 shows the 5 goal levels used for each of the 3 games. The 3 different games that we selected were as follows.

- **Zelda:** is a simple remake of the dungeon system of The Legend of Zelda (Nintendo, 1986). The goal of the game is to get the key and go to the door without dying. The player can either avoid enemies or hit them with their sword for extra points. The goal set levels that we used were all 7×11 tiles. To measure playability, we ensured that there was one key, one door, and one player. Also, the player must've been able to reach the key and the door.
- **Sokoban:** is a japanese puzzle game (Thinking Rabbit, 1982). The goal of the game is to push the crates to a certain target location without getting stuck. The goal levels that we used were of size 5×5 tiles. To measure playability, we needed to have one player and a number of crates equal to the number of targets. Lastly, the A* algorithm must've been able to solve the puzzle in a reasonable amount of time.
- **Danger Dave:** is a simple remake of the PC game Dangerous Dave (John Romero, 1988). This game is different than the other two games as it is a platformer game where the player can move left, right, and jump. The goal of the game is to collect the chalice and then reach the door while avoiding the spikes. The player can collect diamonds to get a higher score. The goal levels used were all of size 7×11 tiles. To measure playability, there must've been one chalice, one door, and one player. Lastly, the A* algorithm must've been able to solve the puzzle in a reasonable amount of time.

For all 3 games, we used a goal set size of 5 because it is a reasonable number of levels that can easily be found in

Observation size	Playable	Playable & Unique
5	$37.98 \pm 0.18\%$	$37.77 \pm 0.1\%$
9	$53.93 \pm 2.82\%$	$28.55 \pm 1.25\%$
15	$86.82 \pm 1.34\%$	$18.34 \pm 1.0\%$

(a) Agent performance by observation size

Goal size	Playable	Playable & Unique
1	$83.84 \pm 3.57\%$	$14.4 \pm 0.26\%$
5	$38.81 \pm 1.85\%$	$28.47 \pm 2.08\%$
50	$37.98 \pm 0.18\%$	$37.77 \pm 0.1\%$

(b) Agent performance by goal size

TABLE I: Agent performance by observation size and goal set size

most games. For both Zelda and Danger Dave, we used an observation size of 5 because both games have the same size map. For Sokoban, we used an observation size of 3 since the map size is much smaller.

D. Different Algorithms

In the last experiment, we benchmark our results against the CESAGAN architecture [14] for Zelda. The authors of this work introduce a bootstrapping training procedure in which a *Conditional Embedding Self-Attention Generative Adversarial Network* is trained to generate Zelda levels using the same goal set of 50 Zelda maps that we used in our experiments. The network performance is evaluated by computing the percentage of playable levels and the percentage of duplicated levels generated. Table III shows the percentage of levels generated that are duplicates. We computed these values to align with the metrics used in the CESAGAN work. We compared the CESGAN baseline to the PoD performance, using a goal set size of 50.

IV. RESULTS

As previously mentioned, for each experiment, we trained 3 networks to ensure stability. Each trained network generated 10,000 levels, which we used to measure agent performance. In the following subsections, we show these performance results.

A. Observation size

Table Ia shows the percentage of playable levels and playable and unique levels generated across different observation sizes. It is evident that the size of the observation is positively correlated to the number of playable levels generated, and negatively correlated to the number of playable and unique levels generated. This is likely due, in part, to the fact that a larger observation size makes it easier for the network to understand and correlate what makes a level playable. However, the larger observation size also makes the network prone to overfitting on the training data. This was expressed as a mode collapse (all levels are similar).

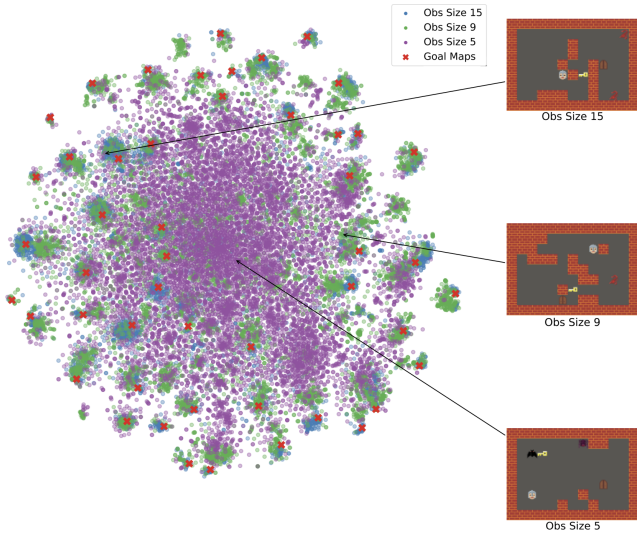


Fig. 5: t-SNE visualization of the unique and playable generated levels for the different trained models based on observation size.

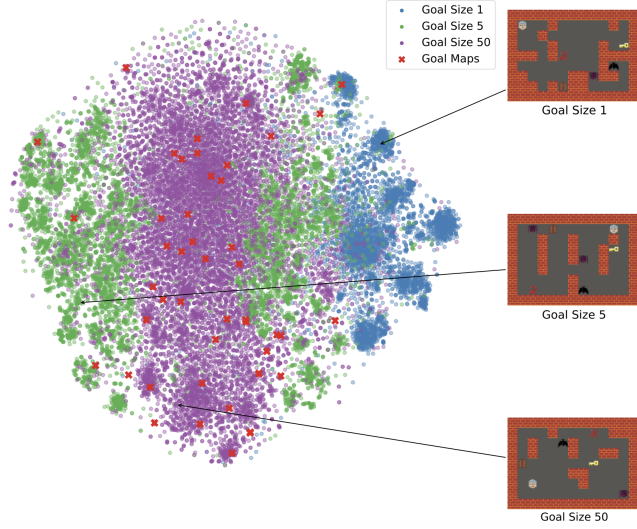


Fig. 6: t-SNE visualization of the generated levels for the different trained models based on using different goal set size.

Figure 5 uses the t-SNE algorithm to plot the generated unique and playable levels for each observation size. The purpose of this visualization is to show the relation of these levels with respect to each other and the goal maps. It is evident that the generated levels from observation size 15 are mostly clustered around the goal maps.

B. Goal set size

Table Ib shows the percentage of playable levels and playable and unique levels generated across different goal set sizes. As is evident in the results, the behavior is similar to what we observed in the observation size experiment. We see that a small goal set size forces the trained generator to overfit on

the training data, resulting in generated levels that are less diverse. By contrast, as the goal set size increases, so to does the diversity of the generated levels. Using a larger goal set size to train the network pushes the network to learn a general strategy that doesn't mode collapse.

From the 2D projection of the generated levels shown in figure 6, we can see that having a larger goal set helps the network to learn to generate levels that are between the goal levels. We noticed that since not all of the 50 levels are very distinct, this could be a limiting factor in the variability of the generated maps. One interesting direction that we did not explore deeply was the mechanism for selecting the goal map in the Path of Destruction method. Since we select the goal maps that are closest to the respective randomly generated starting maps, it is not immediately evident how this selection mechanism affects the diversity of the training set. Furthermore, some goal levels might be more unique than others. Similarly, some goal levels might be more difficult than others. We think the goal level selection method should be investigated in future work.

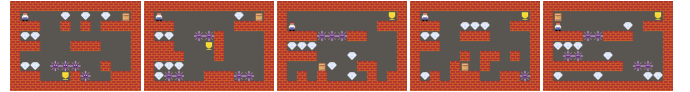
C. Different games



(a) Examples of playable and uniquely generated Zelda levels.



(b) Examples of playable and uniquely generated Sokoban levels.



(c) Examples of playable and uniquely generated Danger Dave levels.

Fig. 7: Examples of playable and uniquely generated levels for Zelda, Sokoban, and Danger Dave.

Figure 7 shows some of the playable and unique levels generated for each of the three games. The levels are sorted from left to right based on how different they are relative to each other and to the goal levels. In comparing these levels to the 5 trained goal levels shown in figure 4, we can see that levels for Sokoban and Danger Dave look more similar to the goal levels than for Zelda. This is also signified in table II, where the percentage of playable and unique levels is much less compared to Zelda. We think that the low diversity in Sokoban is due to the smaller level size which makes the 3x3 observation window more global. This is similar to the observation size of 9 used in Zelda. On the other hand, the Danger Dave agent has low scores for both playability and playability and uniqueness because it is a harder game to generate playable levels for. Incorporating the solution path as part of the goal maps, like in

Game	Playable	Playable & Unique
Zelda	$38.81 \pm 1.85\%$	$28.47 \pm 2.08\%$
Sokoban	$53.31 \pm 0.58\%$	$4.81 \pm 0.02\%$
Danger Dave	$18.2 \pm 0.35\%$	$10.48 \pm 0.02\%$

TABLE II: Agent performance across different games

Model	Playable	Duplicated levels
CESAGAN	47.00%	60.30%
PoD	37.98%	0.00%

TABLE III: Agent performance compared to baseline

the work done using Multidimensional Markov Chains for level generation [15], could improve the overall quality in platformer games.

Although the levels look similar to the goal maps, they have different solutions. This is especially true for Sokoban and Danger Dave because these games are more strategic and thus require more planning compared to Zelda. Since our diversity metric (hamming distance) doesn’t take in account the level solution, we think that a lot of interesting Sokoban levels were subsequently removed. For example, if we change just the target location for one of the puzzles, we could then produce a more complex level that requires many more steps to solve even though the hamming distance will be minimally affected. We think that this is an interesting avenue to explore in future work as it could be used to influence the training data generation steps by selecting goal levels that have high diversity more frequently.

D. Different Algorithms

As is evident from table III, our PoD agent significantly outperformed the baseline in terms of the percentage of duplicate levels generated. The PoD agent did so while remaining competitive in terms of the percentage of playable levels generated. Figure 8 shows examples from PoD and CESAGAN, respectively. The levels generated by the PoD have similar structure to the original levels compared to the CESAGAN levels. CESAGAN levels are more open with less connected tiles. We think that the locality of observation forced the PoD network to learn local patterns such as walls needing to be connected to each other. By contrast, the CESAGAN results seemed to learn a general distribution of the positions of wall tiles. We also noticed that levels generated by CESAGAN tended to have fewer enemies compared to PoD levels. The small number of enemies is closer to the actual number of enemies in the goal levels. We believe that locality is the main driver of this. Having a local view prevents the network from easily learning how to count the number of enemies in the map. The CESAGAN agent, by contrast, learned the distribution of different tiles from the entire map.

V. DISCUSSION

The core underlying idea of the PCG method described here is that generation can be seen as repair. The trained generator



(a) Examples of playable and non-duplicated Zelda levels generated by PoD.



(b) Examples of playable and non-duplicated Zelda levels generated by CESAGAN.

Fig. 8: Examples of playable non-duplicated levels generated by Path of Destruction (PoD) and Conditional Embedding Self-Attention Generative Adversarial Network (CESAGAN), respectively.

is faced with a sequence of broken level segments, and asked to determine the edit that would best repair the broken segment. This is why the levels first need to be “destroyed” in order to train the model: as we already know the correct form of each segment, we can create a dataset of repair actions to learn. Given that any map can be destroyed in a very large number of different ways, one can view the process of dataset generation as a form of data augmentation. This helps to explain the strong results we’ve seen from generation based training on a very small number of levels. While we have not seen the idea of creation as repair expressively stated in computational creativity or creative AI research, it relates to many other ideas. For example, many schools of art teaching focus on how to gradually improve an artifact from an imperfect state.

It is interesting to compare the Path of Destruction method with image inpainting techniques. For centuries, conservators have studied how to best restore art pieces by inpainting parts that are missing or destroyed due to e.g. deterioration or vandalism. In recent years, a burgeoning subfield of AI poses inpainting as a computational problem and advances machine learning-based methods to solve it [16]. These methods typically take a set of images and create training data by randomly removing parts of the images. The model is thus tasked with regenerating the missing parts. In that sense, the overall procedure of image inpainting is similar to the Path of Destruction method presented here. However, in our method, each edit action pertains only to a single tile (or pixel) based on a limited neighboring field of view.

Diffusion Models are another obvious point of comparison³. Those methods are similar to Path of Destruction in that an agent learns a task through repair steps generated from destroyed images [17]. However, there are some differences between PoD and diffusion models. Perhaps the most significant difference is that diffusion models add Gaussian noise to portion of the image in one step. By contrast, in PoD, the level is

³The Path of Destruction technique was developed independently from Diffusion Models, and the similarity was only pointed out to the authors after a first preprint of this paper had been posted.

destroyed iteratively, one tile at a time. Moreover, PoD destroys the level of each episode towards a goal whereas, with diffusion models, the image is destroyed towards a distribution. While diffusion models could be trained on levels instead of images, they normally require extremely large datasets to generalize. Achieving diffusion model generalization using a small dataset of levels constitutes an interesting area of future exploration.

We can also contrast our novel method with PCG methods based on predicting a tile based on immediate neighbor tiles. This includes the widely-used Wave Function Collapse (WFC) algorithm⁴, a constraint-solving algorithm that learns local constraints from a small amount of data and uses them to create new content that recreates local patterns stochastically [18]. It also includes methods based on cellular automata. Specifically, a variation known as Neural Cellular Automata (NCA) can be trained using gradient descent to recreate images [19]. It has also recently been shown that evolutionary methods can be used to create NCAs that create playable levels [13]. Yet another related method is Multidimensional Markov Chains (MdMC), which directly learn the conditional probabilities of neighboring tiles [20]. Compared to these methods, the Path of Destruction takes a larger part of the level into account as part of the input observation. We believe the data augmentation performed by the Path of Destruction method leads to better few-shot generalization compared to methods which learn directly from the source levels.

VI. CONCLUSION

We have presented the Path of Destruction, a procedural content generation method which can train iterative level generators from existing levels. The key innovation is the data generation (or augmentation) method, which makes a large number of small changes to destroy a level and creates a dataset of the destruction played backwards, mapping from a destroyed level to a single repair action. By training convolutional networks on the generated data, we created level generators that reliably produce playable and unique levels for three different 2D games. It is notable that PoD generators can generate a wide variety of levels even when trained on a limited number of goal levels. This was most evident when we measured the PoD agents' performance against the CESAGAN baseline. The results illustrated that the PoD agents significantly outperformed the benchmark in terms of the percentage of unique playable maps generated.

A number of developments of the method proposed here are possible and should be investigated. It is worth further exploring different neural architectures, observation sizes, and representation types. One could also envision using evolutionary search or quality diversity to produce a diverse archive of training episodes to increase the uniqueness of the generated levels.

REFERENCES

- [1] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation

- via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.
- [2] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," in *Procedural Content Generation in Games*. Springer, 2016, pp. 31–55.
- [3] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.
- [4] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011.
- [5] R. Jain, A. Isaksen, C. Holmgård, and J. Togelius, "Autoencoders for level generation, repair, and recognition," in *Proceedings of the ICCG workshop on computational creativity and games*, 2016, p. 9.
- [6] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, "Evolving mario levels in the latent space of a deep convolutional generative adversarial network," in *Proceedings of the genetic and evolutionary computation conference*, 2018, pp. 221–228.
- [7] S. Dahlskog, J. Togelius, and M. J. Nelson, "Linear levels through n-grams," in *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, 2014, pp. 200–206.
- [8] A. J. Summerville and M. Mateas, "Super mario as a string: Platformer level generation via lstms," in *Proceedings of DiGRA/FGD*, 2016.
- [9] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, 2020, pp. 95–101.
- [10] S. Earle, M. Edwards, A. Khalifa, P. Bontrager, and J. Togelius, "Learning controllable content generators," in *2021 IEEE Conference on Games (CoG)*. IEEE, 2021, pp. 1–9.
- [11] O. Delarosa, H. Dong, M. Ruan, A. Khalifa, and J. Togelius, "Mixed-initiative level design with rl brush," in *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*. Springer, 2021, pp. 412–426.
- [12] C. Ye, A. Khalifa, P. Bontrager, and J. Togelius, "Rotation, translation, and cropping for zero-shot generalization," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 57–64.
- [13] S. Earle, J. Snider, M. C. Fontaine, S. Nikolaidis, and J. Togelius, "Illuminating diverse neural cellular automata for level generation," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2022, pp. 68–76.
- [14] R. R. Torrado, A. Khalifa, M. C. Green, N. Justesen, S. Risi, and J. Togelius, "Bootstrapping conditional gans for video game level generation," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 41–48.
- [15] S. Snodgrass and S. Ontanón, "Procedural level generation using multi-layer level representations with mdmcs," in *2017 IEEE conference on computational intelligence and games (CIG)*. IEEE, 2017, pp. 280–287.
- [16] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester, "Image inpainting," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, pp. 417–424.
- [17] P. Dhariwal and A. Nichol, "Diffusion models beat gans on image synthesis," *CoRR*, vol. abs/2105.05233, 2021. [Online]. Available: <https://arxiv.org/abs/2105.05233>
- [18] I. Karth and A. M. Smith, "Wavefunctioncollapse is constraint solving in the wild," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017, pp. 1–10.
- [19] A. Mordvintsev, E. Randazzo, E. Niklasson, and M. Levin, "Growing neural cellular automata," *Distill*, vol. 5, no. 2, p. e23, 2020.
- [20] S. Snodgrass and S. Ontanón, "Controllable procedural content generation via constrained multi-dimensional markov chain sampling," in *IJCAI*, 2016, pp. 780–786.

⁴<https://github.com/mxgmn/WaveFunctionCollapse>