Mutation Models: Learning to Generate Levels by Imitating Evolution

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

Michael Cerny Green
mike.green@nyu.edu
Game Innovation Lab
New York University
Brooklyn, NY, USA

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

ABSTRACT

Search-based procedural content generation (PCG) is a well-known method for level generation in games. Its key advantage is that it is generic and able to satisfy functional constraints. However, due to the heavy computational costs to run these algorithms online, search-based PCG is rarely utilized for real-time generation. In this paper, we introduce mutation models, a new type of iterative level generator based on machine learning. We train a model to imitate the evolutionary process and use the trained model to generate levels. This trained model is able to modify noisy levels sequentially to create better levels without the need for a fitness function during inference. We evaluate our trained models on a 2D maze generation task. We compare several different versions of the method: training the models either at the end of evolution (normal evolution) or every 100 generations (assisted evolution) and using the model as a mutation function during evolution. Using the assisted evolution process, the final trained models are able to generate mazes with a success rate of 99% and high diversity of 86%. The trained model is many times faster than the evolutionary process it was trained on. This work opens the door to a new way of learning level generators guided by an evolutionary process, meaning automatic creation of generators with specifiable constraints and objectives that are fast enough for runtime deployment in games.

KEYWORDS

Neural Networks, Evolution, Data Augmentation, Surrogate Models, Procedural Content Generation, Expressive Range Analysis, Level Generation

1 INTRODUCTION

Very coarsely, we can construct content generators in two different ways. We can either create a generator that constructs an artifact in a finite (often fixed) number of steps without testing during the construction process. The other way is to perform a search or optimization process where either a whole artifact or part of it is tested repeatedly during generation to guide it forward [41]. In general, constructive generators are much faster and therefore better suited to real-time generation than search-based generators. On the other hand, the lack of quality checking during the construction process means that the expressive spaces of the generator may need to be restricted in order to guarantee that the content is playable (not functionally broken). Creating a good constructive generator is hard, and often requires expert human effort for each use case. Generators based on search and optimization, on the other hand, can guarantee playability but are much slower; depending on the cost of content evaluation, they can be unworkably slow.

Could these advantages be combined? Could we create constructive generators that are fast yet have functionality guarantees and a wide expressive range? Could these be created automatically? It stands to reason that one could use machine learning to somehow learn generators. While the computational costs of learning a generator may be large, using the learned model as a constructive generator is computationally cheap; in other words, computation is front-loaded. Various self-supervised approaches have been advanced, building on GANs [42, 43] or autoencoders [18, 34], that learn from existing content to generate new content. Another recent approach, the Path of Destruction, turns existing content into sequences of repair actions that can be modeled [37]. Obviously, self-supervised approaches require a decently-sized library of content, such as levels, to learn from [27, 40].

Alternatively one could use evolution [13, 21, 25] or reinforcement learning [22, 28, 36] (PCGRL) to learn generators. This does not require existing content, but it does necessitate the existence of a good reward or fitness function. Specifying a good fitness/reward function for generating content generators can be much harder than specifying a good fitness/reward function for "simply" generating content. The intuition behind this is that we more or less know what good content looks like, but not necessarily what a good content *generator* looks like. To take a concrete issue, a good fitness/reward function for a generator will probably need to reward appropriately diverse output from the generator.

In this paper, we propose another way of generating competent content generators. Much like PCGRL, we are not reliant on existing content examples. However, our method also does not need a way to evaluate content generator quality; we only need a way to evaluate good content. The goal is to make it possible to create fast content generators that are as reliable, controllable, and easy to specify as search-based generators. In a nutshell, our approach entails building a search-based generator, and training a neural network to take the actions the evolutionary algorithm would take. The sequence of changes that are made by evolution can be seen as training data for a supervised learning process, and the resulting model will take those actions that the evolutionary process would. This can be seen as imitation learning on evolutionary trajectories; the approach has similarities with offline reinforcement and surrogate models in evolution.

2 BACKGROUND

In the following subsections, we review related background work in regards to procedural content generation (PCG), focusing on search-based and machine learning based methods. We also review the concept of surrogate modeling for evolutionary systems.

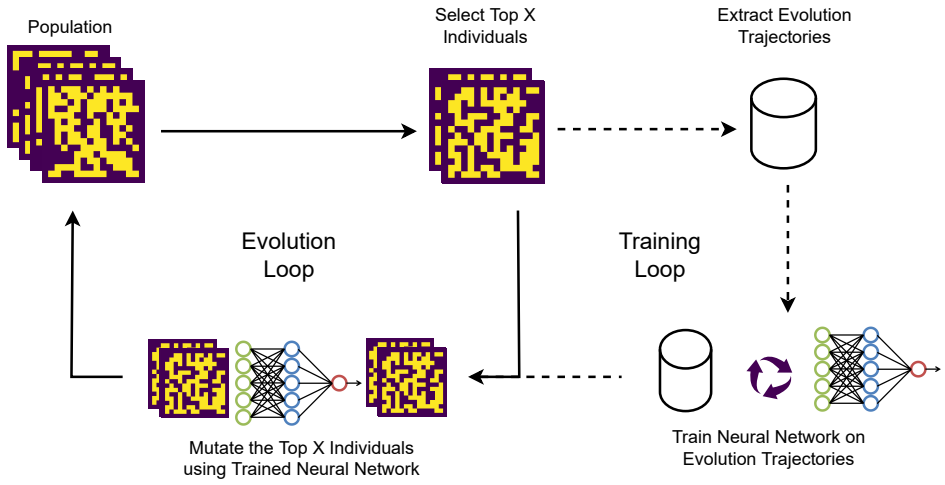


Figure 1: System diagram of the mutation models framework. The system consists of two main parts: evolution loop and training loop. The evolution loop is the usual evolutionary algorithm which evolves chromosomes using a fitness function. The training loop converts the evolution history into a dataset then trains a machine learning model on it. This trained machine learning model can be used to assist evolution by acting as a mutator or only trained at the end of evolution. If the model is being used as a mutator, the training loop happens every I generations to improve the accuracy of the machine learning model. Later when the evolution is done, the trained model can be used as a level generator as it learned to imitate the successful evolution trajectories.

2.1 Search-based PCG

Search-based PCG defines a family of PCG techniques powered by search methods to generate content [41]. Evolutionary algorithms are often used, as they can be applied to many problems like level generation in video games. Search-based PCG has been applied to many frameworks and games, such as the General Video Game AI framework [33], PuzzleScript [23], Cut the Rope (ZeptoLab, 2010) [35], and Mazes [3]. Search-based techniques have been used to generate levels [2, 5, 23], game rules [7, 9, 24] and level generators [13, 21, 25].

2.2 PCGML

Procedural content generation via machine learning (PCGML) is a process of generating content using ML algorithms based on input example. These methods are not often used outside of the research community because of their reliance on large datasets, long training times, and little control of the generated output. There are exceptions: for example, Caves of Qud [15] uses PCGML to generate books and other aesthetic elements. Research applications with PCGML have used many different techniques including Markov Chains [38], N-Grams [10], GANs [42, 43], Autoencoders [18, 34], and LSTMs [39].

Most related to this work is a project by Siper et al. in which a network is trained to repair levels [37], modeling a so-called *Path of Destruction*. This can be done by taking a series of "goal levels" and proceeding to randomly destroy them. A network is then trained on the repair trajectories to learn how to convert these destroyed levels back to the goal levels. The difference between Path of Destruction and most of the recent PCGML work is the network generates the content iteratively (similar to Wave Function Collapse [20] and Markov Random Fields [38]) while for example GAN-based or autoencoder-based generators generate content in one shot (one pass). The approach proposed is similar to Path of Destruction in that we are training a networks to iteratively improve randomized levels, but instead of training it on reversed paths of destruction we train it by imitating the successful mutation operation of evolution.

2.3 PCGRL

Procedural content generation via reinforcement learning (PCGRL) [22] is a process of generating content using RL algorithms. PCGRL transforms the generation process to a game playing process where an agent can take actions at several states and get a reward based on that. The work proposes three different methods to achieve that: narrow, turtle, and wide representations. Narrow representation transforms the generation into a process of asking the agent about each tile in the level if it needs to change or not and if yes what is the value of change. On the other hand, turtle representation has more control over the location by allowing the agent to either modify the current tile (like narrow) or move to a neighboring tile. Finally, wide representation provides the agent with maximum control by allowing the agent to select the location freely and the modification value for it. In this work, we will be using the narrow representation as it provides a small action space for the agent and has comparable results to the other two [22].

PCGRL techniques have been used in level/experience generation [8, 12, 28, 31, 36, 44, 46]. It is difficult to transform content generation into a reinforcement environment, thus there is a few PCGRL examples in either academia or industry compared to other methods. Similar to Path of Destruction, PCGRL generators are

iterative, meaning that they produce content using multiple modification steps, action by action. This grants certain advantages to PCGRL: for example, it is often easier to build mixed-initiative systems around them [11, 16, 17].

2.4 Surrogate Modeling

Surrogate models are models of computationally expensive processes, which can then be used in lieu of the process itself [32]. Surrogate models are meant to be easier to evaluate than the process it models, which translates to time and computational improvements. Surrogates are typically constructed using a data-driven, bottom-up approach, typically by training a network on a distribution of intelligently selected data points.

Within evolutionary computation, it is common to use machine learning to build surrogate models of the fitness function [19]. A fitness surrogate model takes a genome as input and returns a fitness value, just as the fitness function would; the advantage is that the trained model is much faster than the actual fitness function. In evolutionary algorithm applications where the fitness function commonly dominates the computation time, surrogate modeling is very useful. These surrogate models can be trained during an individual evolutionary run or over several such runs. While it would be possible in principle to learn surrogate models of all parts of an evolutionary algorithm, they are predominantly applied to the fitness function. The only attempt we know to create surrogate-based mutation is built on a surrogate of the fitness function rather than the mutation function itself [1].

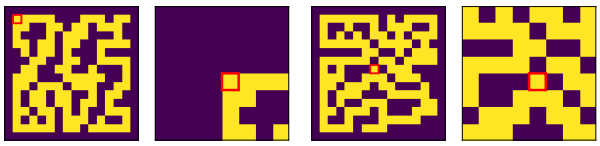
3 MUTATION MODELS FRAMEWORK

In this paper, we train a machine learning model to imitate the evolutionary process. The model takes a chromosome (in this case, level) as input and outputs a modification to it. The end result is a model that can generate content without the need for an evaluation function. Figure 1 shows the full system diagram of the mutation models framework, which consists of two parts: the evolution loop and the training loop. The evolution loop is a standard evolutionary algorithm which evolves content by trying to increase fitness. Within the training loop, data is extracted from the evolution loop to train a machine learning model to perform successful mutations (mutations that leads to higher fitness) like the evolutionary process, thus the name *mutation models*. The next two subsections explain both loops in detail.

3.1 Evolution Loop

The evolution loop is a normal, simple evolutionary algorithm. In this work, we use the standard $\mu + \lambda$ evolution strategy without self-adaptation [4]. The process as shown in Figure 1 is simple: it optimizes levels to maximize their fitness. Unlike most evolutionary methods, our method retains the evolution history of every chromosome by recording the process of mutations made to get to its current form. The following steps explain the evolution loop used in this work:

(1) Generate a random population of levels of size $\mu + \lambda$.
(2) Pick the top μ levels based on the fitness function to be parents for the next generation.



(a) Mutation location (x=0, y=0) (b) Mutation Location (x=6, y=6)

Figure 2: Examples of transforming a level and a mutation location to a cropped $8x8$ level. This $8x8$ cropped level act as the input for our trained machine learning model.

(3) Generate λ new levels by mutating the parent chromosomes (μ levels).
(4) Repeat the above steps for G generations or until convergence.

We are not restricted to only use a $\mu + \lambda$ evolution strategy algorithm. Other evolutionary algorithms would work well as long as they do not require crossover and allow us to keep track of the evolutionary history of the chromosomes across generations (for example, in a continuous domain it would straightforward to use the high-performing CMA-ES algorithm). However, the selected mutation function will determine how the machine learning algorithm imitates evolution. In this work, the mutations are small, they only change one tile at a time. For example: the system picks a random x,y coordinates (location) in the level and swaps the current tile value with a new tile value or decide not change at all (action). The action does not always need to be random: this could be sampled from a machine learning model trained to perform successful mutations (mutations that improve the fitness function). This will be discussed in the next subsection.

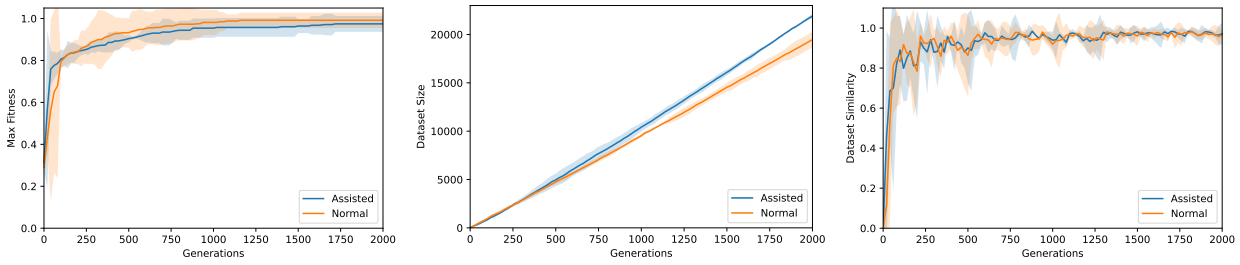
3.2 Training Loop

The training loop is responsible for training a machine learning algorithm to learn the evolutionary process. The idea is to train a model on the successful mutations which cause the top X chromosomes from the population to have a high fitness. Doing so allows the model to learn mutations that are likely to lead to improvement of the level's fitness. The following steps explain the training loop used in this work:

(1) Pick top X chromosomes from the population.
(2) Extract the trajectory data (evolution history) from these chromosomes to build a dataset of levels and mutations.
(3) Train a machine learning model on the extracted dataset.

The network's training loop can occur at the end of evolution on the top X evolved maps, which we call the "Normal" method. It can also occur every I generations, allowing the network to be used as the mutation operator during the evolution loop, which we call the "Assisted" method. The "Assisted" method is similar in idea to on-policy reinforcement learning: a network creates its own dataset, which it then trains on. In contrast to traditional reinforcement learning, the evolutionary process here is helping filter the training dataset by only keeping the best data points.

In order to build the training dataset, we need to convert the evolution history of a chromosome into inputs and outputs for the machine learning model. We found inspiration in the "Narrow"



(a) Fitness over time **(b) Dataset size over time** **(c) Evolution history similarity over time**

Figure 3: Fitness, dataset size, and dataset similarity of the top 10 chromosomes of each generation over the course of "Normal" evolution and "Assisted" evolution. The shaded area shows the 95% confidence interval over 3 runs.

representation in the PCGRL [22] environment. In the narrow representation, the input is the mutation location and level state at a certain point in history, while the output is the mutation value (no change or the new tile value). In preliminary experiments, we realized that using the entire level state does not help the machine learning model generalize. We follow the work by Siper et al. [37] and Ye et al. [45] in cropping the level state around the mutation location, as shown in Figure 2.

4 EXPERIMENTS

To demonstrate each method's capabilities, we apply them on a simple domain in which they generate 2D maze layouts. We test the two methods of training explained in section 3.2: "Normal" and "Assisted". The "Normal" method trains the model at the end after evolution is done, while "Assisted" method retrains the model from scratch every 100 generations and uses the freshly-trained model as a mutation function in the evolution loop. All our experiments are repeated 3 times (the full evolution loop including training a neural network either at the end of evolution in the case of "Normal" or during evolution in the case of "Assisted"), and we show the average max fitness, dataset size, and dataset similarity across these runs and the 95% confidence interval. After evolution is done, the final trained models from each method are tested for their capabilities on generating content without the need for a fitness function.

4.1 Domain

We use a 2D maze layout of size $14x14$ (excluding the borders) as our proof-of-concept test-bed. The goal is to generate a 2D layout of solid and empty tiles such that all the empty tiles are accessible from any other empty tile and the longest path in that layout is maximized. Figure 4 shows the top 4 evolved maps by the evolutionary algorithm showcasing full connectivity and long winding paths.

4.2 Evolution Loop

As discussed in section 3.1, we are using $\mu + \lambda$ evolution strategy with $\mu = \lambda = 50$. The starting levels are initialized using a uniform distribution with 50% probability for each tile to be either empty or solid. The mutation operator picks a random location (x and y position) and uses either a random mutation value for the "Normal" method, or samples the value from the trained machine learning model for the "Assisted" method. To guarantee diversity in the "Assisted" method, there is a 25% chance to sample a random action instead from the trained model. We run the evolution process for 2000 generations with a cascading fitness function which tries to satisfy the connectivity constraint first then maximizes path length as shown in equation 1:

$$f = \begin{cases} 0.5 \cdot (1 - \frac{n}{20}) & \text{if } n > 1 \text{ and } n < 20 \\ 0.5 + 0.5 \cdot \frac{p}{98} & \text{if } n == 1 \\ 0 & otherwise \end{cases} \quad (1)$$

where n is the number of regions in the current level and p is the length of the longest path length in the map.

4.3 Training Loop

The training loop is responsible about two things, creating a training dataset for the model and training the model. For the dataset generation, we pick the top 10 chromosomes from the population and extract their evolution history. The evolution history consists of the current level, the mutation location, and the mutation action at every generation in a chromosome's history. We transform the level and mutation location into an single image by adopting the narrow representation from the PCGRL [22] framework. We crop the level around the mutation location to be of size $8x8$ as shown in Figure 2. This cropped level is used as the input for the machine learning model. The output of the model is the mutation action that corresponds to the cropped level. In this domain, there are 3 different mutation actions: "No Change", "Change to Empty", and "Change to Solid".

The machine learning model is a small convolutional neural network (224,771 parameters) with a similar architecture to the Atari Deep-Q network [29]. The network consists of 3 convolutional layers (with $3x3$ filters and kernel size of 32, 64, and 128 respectively) followed by 2 fully connected layers (with 256 and 3 neurons respectively). We use 2 max pooling layers, each size 2, after the first and second convolution layers to decrease the input space size. All convolutional layers use same padding to make sure the size of the input remains constant. All activation functions are Relu except for the last layer, where we use softmax. The neural network is trained using an Adam optimizer with learning rate of 10^{-4}, a minibatch size of 32, and categorical cross entropy loss. For the "Normal" method, we train a new network for 2, 4, and

8 epochs to explore how different epochs impact network results. For the "Assisted" method, we only train the network for 2 epochs due to the computation costs. The "Assisted" method network is retrained from scratch (with random weights and a reset learning rate) every 100 generations. This decision was decided as preliminary experiments showed that models trained continuously had a higher chance to overfit on earlier data.

4.4 Model Inference

During the evolution/training step, both the "Normal" and "Assisted" methods utilize a fitness function. The evolutionary algorithm creates a dataset which is used to train a neural network to imitate the evolutionary process. After the model is trained (during *inference*), the trained model does **not** use any fitness function in **either** method. The inference step is the same regardless of which evolution/training procedure is used. We use the final trained network from each experiment to generate 100 levels starting from noise (50% solid and 50% empty levels). The network actions are sampled using the probability distribution from the last softmax layer. The trained networks are run sequentially like a scanline over the entire map multiple times, as proposed in [22]. The networks stop iterating if they reach one of two cases: success (all the empty tiles are fully connected) or failure (iterated on the whole map for 196 times). We record the success rate and the iteration number that the model requires to reach a successful level. We also record the path length of each of the successful generated levels with the number of empty tiles, which we use to calculate the diversity of the generated content. This diversity is shown both as an expressive range analysis as well as a percentage of generated levels having different combinations of path lengths and empty tiles.

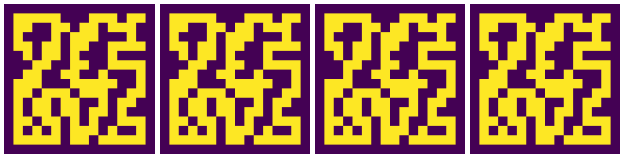
5 RESULTS

In this section, we explore the results from our system. We start showing the evolution results in Section 5.1. These results are collected from the evolutionary generator which uses an evaluation function to create levels. We analyze the effect of using the "Normal" vs "Assisted" methods on the evolution loop. In contrast, Section 5.2 shows the results of the model inference. The results are collected from the final trained models, which do not use a fitness function during inference.

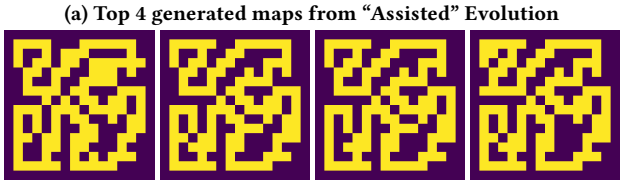
5.1 Evolution Results

Figure 3 displays different evolution metrics across the 2000 generations. The usage of "Assisted" method does not change the fitness performance of the evolutionary algorithm with respect to the "Normal" method. The only noticeable difference is the size of the extracted dataset (Figure 3b) where the "Assisted" method manages to create a slightly larger dataset than the "Normal" method. This could be an effect of the network refusing to change some chromosomes early (using the"No Change" action) when the mutation location is not good enough. This would lead to these chromosomes having a longer history than usual as they survive for multiple generations.

The last metric we observe is the similarities between the evolution histories of the top 10 chromosomes. We notice that the top 10 begin with very different trajectories but end up having almost identical trajectories with only slight differences (Figure 3c). This was expected as evolutionary strategy tends to converge when most chromosomes in the population become highly similar to each other as shown in Figure 4. In future work, this might be solved by using a more complex evolutionary algorithm or a quality diversity algorithm such as MAP-Elites [30] which could ensure diverse trajectories. The similarity metric that we use is based on the Ratcliff-Obershelp algorithm [6].



(a) Top 4 generated maps from "Assisted" Evolution



(b) Top 4 generated maps from "Normal" Evolution

Figure 4: Top four maps at generation 2000, from two different evolutionary runs ("Assisted" evolution and "Normal" evolution). All the four maps look almost identical with few tile differences as evolutionary strategy tends to converge when most chromosomes in the population become highly similar to each other.

Method	Success	Diversity	Iterations
Assisted-2	**99.67% ± 0.49%**	**86.83% ± 3.8%**	**18.21 ± 18.57**
Normal-2	30.17% ± 32.7%	28.5% ± 30.62%	61.7 ± 47.22
Normal-4	66.83% ± 49.22%	59.33% ± 43.69%	23.55 ± 48.59
Normal-8	65.17% ± 48.37%	60% ± 44.59%	31.78 ± 23.84

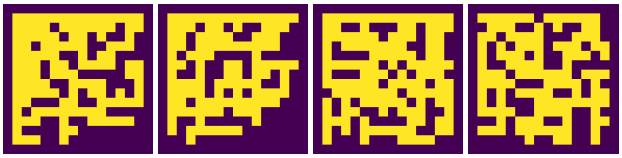
Table 1: The average and 95% confidence interval of 100 generated levels from 3 runs for success rate, diversity, and number of iterations by the final trained network from "Assisted" and "Normal" evolution. The "-#" after the name of an experiment designates the number of epochs the final network used for training.

5.2 Model Inference Results

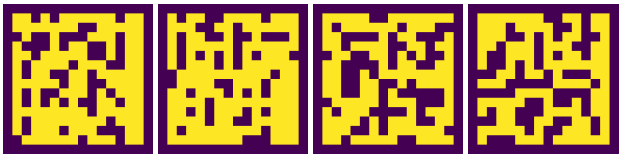
In this section, we compare the results between the final trained network from the "Assisted" method using 2 epochs ($Assisted-2$) and the final trained networks from the "Normal" method using 2, 4, and 8 epochs ($Normal-2$, $Normal-4$, and $Normal-8$ respectively). Table 1 displays the success rate and diversity of the generated levels as well as the number of iterations each model needs to reach success (which we will refer from now on as "iterations"). The diversity and iterations are only calculated on the successfully generated levels, and not for failures.

All of the "Normal" trained models appear sensitive to the extracted dataset and varied widely in performance and diversity. We believe that using random distributions of mutations in the "Normal"

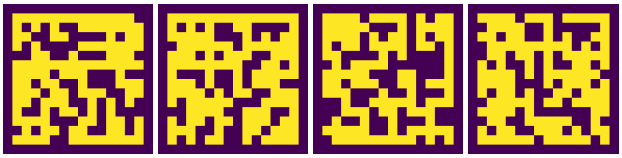
(a) Assisted-2



(b) Normal-2



(c) Normal-4



(d) Normal-8

Figure 5: Four successful maps generated from the different trained model via inference. None of these maps are generated using a fitness function. Each subcaption specifies the name of the experiment, divided into 2 parts: name and number. The name reflects the evolution process that created the dataset for this model to be trained on ("Normal" or "Assisted"). The number after the name of the experiment designates the number of epochs the network used for training.

method to build the evolution trajectories produces a noisy and unstable dataset to train on. However, trajectories built using the output of the trained network used by the "Assisted" method seem to produce a more consistent dataset that is easier to train a policy on. This is similar to on-policy reinforcement learning where the network is trying to learn from it own experience using rewards. In this work, the reward comes in the form of dataset filtering through evolution: only successful mutations are kept, and these mutations are the ones that the network will be trained on. It is not surprising that training for more epochs improves the performance of "Normal" method networks. However, this performance plateaus after only 4 epochs which confirms that the "Assisted" method helps training more than we expected. All of the networks take many iterations to converge on a successful level with minimum average of 18 and maximum average of 61. We believe that the "Assisted-2" may take less iterations if it is trained for more epochs with a result similar to what happens for the "Normal" methods.

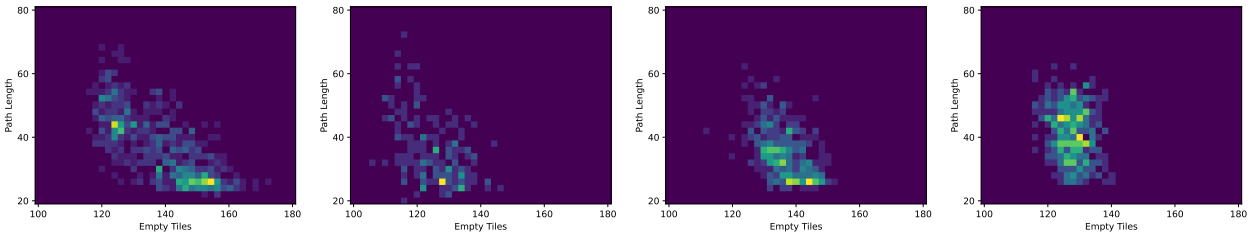
Figure 5 displays four successful maps from each experiment. The "Assisted" generated maps (Figure 5a), have features similar to the original final maps. They contain long paths which contain long vertical lines. The "Normal-2" generated maps are more open in the upper left corner. We believe the small amount of epochs allows the network to learn a very simple policy which erases tiles to have full connectivity. "Normal-4" and "Normal-8" models also have more open space compared to "Assisted" but appeared more structured than "Normal-2". Often, the easiest way to reach full connectivity is to remove tiles from the level, as this will naturally "open it up." We believe that "Normal-2" levels appear more open is because of the cascaded fitness function, where the evolution trajectory starts by connecting all the empty tiles then later improve path length. These evolution trajectories tend to have more "Change to Empty" actions at the beginning of evolution. We believe that the trained policy might have learned this behavior when the level is noisy. It learns to fully connect the map first, then later increase path length. This doesn't happen in the "Assisted-2" maps since the evolution is guided by a neural network that can take different actions which can be adjusted later. And since we terminate generation when everything is connected, "Normal" networks might achieve full connectivity just as generation ends, and before it begins to add solid tiles, giving the feeling of open space.

Finally, Figure 6 displays the expressive range analysis of the trained models across 3 runs. Although "Assisted-2" generated maps have winding paths, they use on average more empty tiles compared to the rest of the models. Although it appears that the "Normal" method uses more empty tiles due to the openness of the maps, this is not always the case. For example, "Normal-2" maps seems more empty but a lot of the bottom left corner is solid. This is the same with "Normal-4" and "Normal-8" where most of the walls are clustered in the center. "Normal" maps give the feeling of openness while containing more solid tiles compared to the "Assisted" generated maps.

6 DISCUSSION

We can contrast the mutation models method proposed in this paper to surrogate-assisted evolutionary computation. From a reinforcement learning perspective, fitness surrogate models can be thought of as models of the V-function, or in other words state value estimators. By using such a model to test multiple actions (mutations/crossover) the best action (surviving genome) can be chosen. The mutation models learned here are instead models of direct action selection. They are similar to models of the Q-function (action value estimators), but instead of returning the value of a particular mutation, they simply return the mutation to make (similar to actor networks).

Imitating mutation may not have the same "speedup effect" on evolution as surrogate modeling fitness functions as we still use the fitness function during evolution (which is computationally intensive) and the fitness value increases in the same rate as normal evolution (according to figure 3a). However, they can provide expressive range where constructive generation methods fail to do so. Figure 6 shows that the "Assisted-2" method especially has a lot of promise in this area. Table 1 shows that the "Assisted-2" results in the highest map diversity. We use the top 10 chromosomes of each generation to train with. Perhaps by expanding this, we might find even greater expressive range.



(a) Assisted-2 (b) Normal-2 (c) Normal-4 (d) Normal-8

Figure 6: Expressive range analysis of the generated maps during inference using the trained networks from the 4 different experiments. The expressive range plots all the successful generated maps on a 2D space, organized by the number of empty tiles and longest path length. Like Figure 5, the name of experiment is divided into 2 parts: name and number. The name reflects the evolution process that created the dataset for this model to be trained on ("Normal" or "Assisted"). The 'number after the name of the experiment designates the number of epochs the network used for training.

Method	Success	Diversity	Wall Time (secs)
Network	99.67% ± 0.49%	86.83% ± 3.8%	**0.6612 ± 2.3874**
Evolution	**100%**	**96%**	12.6957 ± 2.2571

Table 2: The average and 95% confidence interval of 100 generated levels by 3 networks trained using "Assisted" evolution for success rate, diversity, and wall clock time compared to the average for success rate, diversity, and wall clock time of running 100 normal evolutionary runs till the evolution find a fully connected level.

Search-based PCG methods are computationally expensive and are thus not commonly used for online generation. We ran an experiment to measure the wall clock time of normal evolution till it find a fully connected level or 2000 generations reached. The evolution took roughly 20x longer than mutation model inferencing as seen in table 2. This difference would be even greater if we had used a more expensive fitness function during evolution, such as gameplay simulation. In other words, the speed advantage of Mutation Models will increase as the generation task gets harder. Learning generators negates the need for an evaluation function like fitness, thus speeding up the overall generation process during inference. Evaluation is what typically takes most of the computational time for search-based generators. Through the use of our method, developers might be able to benefit from the advantages that search-based PCG provides without having to bear the cost of online computation. Although evolution guarantees 100% success rate and almost 100% diversity as seen in table 2, the "Assisted" trained networks have comparable success rate and diversity with drastically less time needed to generate the content.

We restricted the evolutionary algorithm to only have mutation to simplify the process of extracting trajectories. Removing crossover usually hinders evolution and in some problems the evolution might get stuck in local optima. This was not the case in our problem due to its simplicity. There is multiple different methods to extract trajectories while crossover is happening. These methods can be categorised into two main directions:

- Converting the crossover into small mutation steps. When we extract trajectories whenever a crossover switch a big amount of tiles, we compare these tiles to the state before crossover and unpack these changes one a time in a random order (similar to path of destruction [37]).
- Consider mutation as a new starting point for a new trajectory. When we extract trajectories whenever a crossover happen we consider the current game level as new level and there is no dependency on the previous level.

This work introduces the opportunity to use an evolutionary algorithm as an assisted process for reinforcement learning, similar to Go-Explore algorithm [14] and offline reinforcement learning [26]. There is a lot to be explored and studied about this new paradigm. For example, what will happen if the evolution trajectories between the selected chromosomes are different? If we use a quality diversity algorithm such as MAP-Elites [30], can we possibly learn a conditional mutator that can change the content towards a certain area in the generative space? We only discussed a very simple test bed problem, do the results from this test bed generalize to more complex games or even generic optimization problems? Also, what about the training algorithm, we only used basic supervised learning on trajectories but one could try using Backward algorithm similar to the one used in Go-Explore to get better results [14]. Finally, we only explored creating the dataset from the evolution history, what if we can create the dataset using a similar technique to the Path of Destruction [37] where the evolution process is used as a method to generate the goal set. We believe that using something other than evolutionary history can free us to use more advanced evolutionary methods and/or indirect representation.

7 CONCLUSION

This paper proposes Mutation Models, a new method of generating content by building a machine learning model that imitates evolution. We use evolution trajectories to train a machine learning models in order to imitate mutation. This allows us to automatically create content generators by only specifying the fitness of the content, which is relatively domain-agnostic. The end result is an iterative content generator that is fast and does not need a fitness function during inference. Model building consists of two main loops (as shown in Figure 1): the evolution loop and training loop. The evolution loop is a normal evolutionary algorithm that generates content, while the training loop is responsible for extracting a

training dataset from the top X chromosomes. The training dataset is constructed by converting the evolution history into state-action pairs, on which a traditional supervised learning model is trained. We propose two methods for the framework: the "Normal" method (the machine learning model is trained after the evolution ends) and the "Assisted" method (the machine learning model is trained every I generations and used to assist evolution). The results show that although the "Assisted" method does not help the evolution to be faster or more efficient, it does stabilize the trained networks, which learn better policies to imitate evolution in comparison to "Normal" method. Both methods can infer on randomly initialized maps and repair them without use of an evaluation function.

REFERENCES

[1] Kamal Abboud and Marc Schoenauer. 2001. Surrogate deterministic mutation: Preliminary results. In *International Conference on Artificial Evolution*. Springer, 104–116.

[2] Daniel Ashlock. 2010. Automatic generation of game elements via evolution. In *Conference on Computational Intelligence and Games*. IEEE, 289–296.

[3] Daniel Ashlock, Colin Lee, and Cameron McGuinness. 2011. Search-based procedural generation of maze-like levels. *Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 260–273.

[4] Hans-Georg Beyer and Hans-Paul Schwefel. 2002. Evolution strategies–a comprehensive introduction. *Natural computing* 1, 1 (2002), 3–52.

[5] Debosmita Bhaumik, Ahmed Khalifa, Michael Green, and Julian Togelius. 2020. Tree search versus optimization approaches for map generation. In *Artificial Intelligence and Interactive Digital Entertainment*, Vol. 16. AAAI, 24–30.

[6] Paul E. Black. 2021. *Ratcliff/Obershelp pattern recognition.* https://www.nist.gov/dads/HTML/ratcliffObershelp.html

[7] Cameron Browne and Frederic Maire. 2010. Evolutionary game design. *Transactions on Computational Intelligence and AI in Games* 2, 1 (2010), 1–16.

[8] Zhengxing Chen, Christopher Amato, Truong-Huy D Nguyen, Seth Cooper, Yizhou Sun, and Magy Seif El-Nasr. 2018. Q-deckrec: A fast deck recommendation system for collectible card games. In *Computational Intelligence and Games*. IEEE, 1–8.

[9] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *European Conference on the Applications of Evolutionary Computation*. Springer, 284–293.

[10] Steve Dahlskog, Julian Togelius, and Mark J Nelson. 2014. Linear levels through n-grams. In *International Academic MindTrek Conference: Media Business, Management, Content & Services*. ACM, 200–206.

[11] Omar Delarosa, Hang Dong, Mindy Ruan, Ahmed Khalifa, and Julian Togelius. 2021. Mixed-initiative level design with rl brush. In *Conference on Computational Intelligence in Music, Sound, Art and Design*. Springer, 412–426.

[12] Sam Earle, Maria Edwards, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. 2021. Learning controllable content generators. In *Conference on Games*. IEEE, 1–9.

[13] Sam Earle, Justin Snider, Matthew C Fontaine, Stefanos Nikolaidis, and Julian Togelius. 2022. Illuminating diverse neural cellular automata for level generation. In *Genetic and Evolutionary Computation Conference*. ACM, 68–76.

[14] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. 2019. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995* (2019).

[15] Jason Grinblat. 2016. *Markov by candlelight.* https://www.youtube.com/watch?v=3AjlsTtrfVY

[16] Matthew Guzdial, Nicholas Liao, Jonathan Chen, Shao-Yu Chen, Shukan Shah, Vishwa Shah, Joshua Reno, Gillian Smith, and Mark O Riedl. 2019. Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators. In *CHI conference on human factors in computing systems*. ACM, 1–13.

[17] Matthew Guzdial, Nicholas Liao, and Mark Riedl. 2018. Co-creative level design via machine learning. In *AIIDE workshop on Experimental AI in Games*. AAAI.

[18] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. 2016. Autoencoders for level generation, repair, and recognition. In *ICCC workshop on computational creativity and games*, Vol. 9. IEEE.

[19] Yaochu Jin. 2011. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation* 1, 2 (2011), 61–70.

[20] Isaac Karth and Adam M Smith. 2017. WaveFunctionCollapse is constraint solving in the wild. In *Foundations of Digital Games*. ACM, 1–10.

[21] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N Yannakakis. 2012. A procedural procedural level generator generator. In *Computational Intelligence and Games*. IEEE, 335–341.

[22] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. Pcgrl: Procedural content generation via reinforcement learning. In *Artificial Intelligence and Interactive Digital Entertainment*, Vol. 16. AAAI, 95–101.

[23] Ahmed Khalifa and Magda Fayek. 2015. Automatic puzzle level generation: A general approach using a description language. In *ICCC workshop on computational creativity and games*. IEEE.

[24] Ahmed Khalifa, Michael Cerny Green, Diego Perez-Liebana, and Julian Togelius. 2017. General video game rule generation. In *Computational Intelligence and Games*. IEEE, 170–177.

[25] Ahmed Khalifa and Julian Togelius. 2020. Multi-Objective level generator generation with Marahel. In *Foundations of Digital Games*. ACM, 1–8.

[26] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643* (2020).

[27] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N Yannakakis, and Julian Togelius. 2021. Deep learning for procedural content generation. *Neural Computing and Applications* 33, 1 (2021), 19–37.

[28] Athar Mahmoudi-Nejad, Matthew Guzdial, and Pierre Boulanger. 2021. Arachnophobia exposure therapy using experience-driven procedural content generation via reinforcement learning (EDPCGRL). In *Artificial Intelligence and Interactive Digital Entertainment*, Vol. 17. AAAI, 164–171.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[30] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).

[31] SangGyu Nam and Kokolo Ikeda. 2019. Generation of diverse stages in turn-based role-playing game using reinforcement learning. In *Conference on Games*. IEEE, 1–8.

[32] Yew S Ong, Prasanth B Nair, and Andrew J Keane. 2003. Evolutionary optimization of computationally expensive problems via surrogate modeling. *AIAA journal* 41, 4 (2003), 687–696.

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

[34] Anurag Sarkar and Seth Cooper. 2021. Dungeon and Platformer Level Blending and Generation using Conditional VAEs. In *Conference on Games*. IEEE, 1–8.

[35] Noor Shaker, Mohammad Shaker, and Julian Togelius. 2013. Evolving playable content for cut the rope through a simulation-based approach. In *Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI.

[36] Tianye Shu, Jialin Liu, and Georgios N Yannakakis. 2021. Experience-driven PCG via reinforcement learning: A Super Mario Bros study. In *Conference on Games*. IEEE, 1–9.

[37] Matthew Siper, Ahmed Khalifa, and Julian Togelius. 2022. Path of Destruction: Learning an Iterative Level Generator Using a Small Dataset. *arXiv preprint arXiv:2202.10184* (2022).

[38] Sam Snodgrass and Santiago Ontanón. 2016. Learning to generate video game maps using markov models. *Transactions on computational intelligence and AI in games* 9, 4 (2016), 410–422.

[39] Adam Summerville, Matthew Guzdial, Michael Mateas, and Mark O Riedl. 2016. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *Artificial intelligence and interactive digital entertainment conference*. AAAI.

[40] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural content generation via machine learning (PCGML). *Transactions on Games* 10, 3 (2018), 257–270.

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

[42] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2020. Bootstrapping conditional gans for video game level generation. In *Conference on Games*. IEEE, 41–48.

[43] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Genetic and evolutionary computation conference*. ACM, 221–228.

[44] Mariana Werneck and Esteban WG Clua. 2020. Generating procedural dungeons using machine learning methods. In *Brazilian Symposium on Computer Games and Digital Entertainment*. IEEE, 90–96.

[45] Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. 2020. Rotation, translation, and cropping for zero-shot generalization. In *Conference on Games*. IEEE, 57–64.

[46] Yahia Zakaria, Magda Fayek, and Mayada Hadhoud. 2022. Procedural Level Generation for Sokoban via Deep Learning: An Experimental Study. *Transactions on Games* (2022).