CPPN2GAN: Combining Compositional Pattern Producing Networks and GANs for Large-scale Pattern Generation

Jacob Schrum
Southwestern University
Georgetown, Texas, USA
schrum2@southwestern.edu

Vanessa Volz
modl.ai
Copenhagen, Denmark
vanessa@modl.ai

Sebastian Risi
modl.ai, IT University of Copenhagen
Copenhagen, Denmark
sebastian@modl.ai

ABSTRACT

Generative Adversarial Networks (GANs) are proving to be a powerful indirect genotype-to-phenotype mapping for evolutionary search, but they have limitations. In particular, GAN output does not scale to arbitrary dimensions, and there is no obvious way of combining multiple GAN outputs into a cohesive whole, which would be useful in many areas, such as the generation of video game levels. Game levels often consist of several segments, sometimes repeated directly or with variation, organized into an engaging pattern. Such patterns can be produced with Compositional Pattern Producing Networks (CPPNs). Specifically, a CPPN can define latent vector GAN inputs as a function of geometry, which provides a way to organize level segments output by a GAN into a complete level. This new CPPN2GAN approach is validated in both *Super Mario Bros.* and *The Legend of Zelda*. Specifically, divergent search via MAP-Elites demonstrates that CPPN2GAN can better cover the space of possible levels. The layouts of the resulting levels are also more cohesive and aesthetically consistent.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Generative and developmental approaches**; **Learning latent representations**; **Genetic algorithms**;

KEYWORDS

Compositional Pattern Producing Networks, Generative Adversarial Networks, Indirect Encoding, Neuroevolution

1 INTRODUCTION

Generative Adversarial Networks (GANs [11]), which are a type of generative neural network that are trained in an unsupervised way, have been shown capable of reproducing certain aspects of a given

training set. For example, they can generate diverse high-resolution samples of a variety of different image classes [4].

However, it is an open question how such a GAN-based approach can scale to generate arbitrarily large artefacts that have a modular structure, such as complete game levels. Game levels can consist of many segments repeated many times, often with variation. Several recent works have shown that it is possible to learn the structure of video game levels using GANs [12, 18, 25, 29], but these approaches only generate small level segments.

In particular, Volz et al. [29] trained a GAN on small segments of Super Mario Bros. levels and searched the induced latent space with CMA-ES [14] to find segments with different properties. In effect, the GAN learned a compact and robust *genotype-to-phenotype* mapping, in which most generated artefacts resemble valid level segments. Training such a GAN on complete Mario levels instead of segments would be challenging due to the drastic reduction in available samples, and the large variation in scale across different levels. While some GAN-based approaches exist to generate complete levels for e.g. the game DOOM [10], they only encode the high-level map structure without low-level details. Instead, this paper aims to generate complete levels with low-level details, by generating multiple segments with a GAN, and combining them into a cohesive global pattern. For example, different segments might increase smoothly in difficulty, have repeating structures, or certain motifs might appear in a symmetric fashion.

This paper builds on a method that has proven its ability to generate patterns with regularities such as symmetry, repetition, and repetition with variation: Compositional Pattern Producing Networks (CPPNs [21]). A CPPN is a special type of neural network that compactly describes patterns with regularities and has succeeded in a variety of domains [5, 7, 15, 16, 19, 20, 26]. The main contribution of this paper is a way of combining one GAN and one CPPN: evolve a CPPN that takes Cartesian coordinates of level segments as input, and produces latent vectors for a GAN that was pre-trained on existing level content (Fig. 1). This combination of technologies can generate large-scale and regular video game maps.

Experiments in this paper apply the new technique, CPPN2GAN, to the games Super Mario Bros. and The Legend of Zelda[1]. Data from the Video Game Level Corpus [24] is used to train GAN models for each game, and CPPNs are evolved using these GANs. CPPN2GAN is compared to the state-of-the-art for GAN-based level generation, which is the evolution of real-valued genomes consisting of multiple GAN latent vectors, one per segment. This control method is called Direct2GAN. Both approaches use the Quality Diversity algorithm MAP-Elites [17] to cover the space of possible level designs. Results below show that Direct2GAN is significantly inferior to the new

---

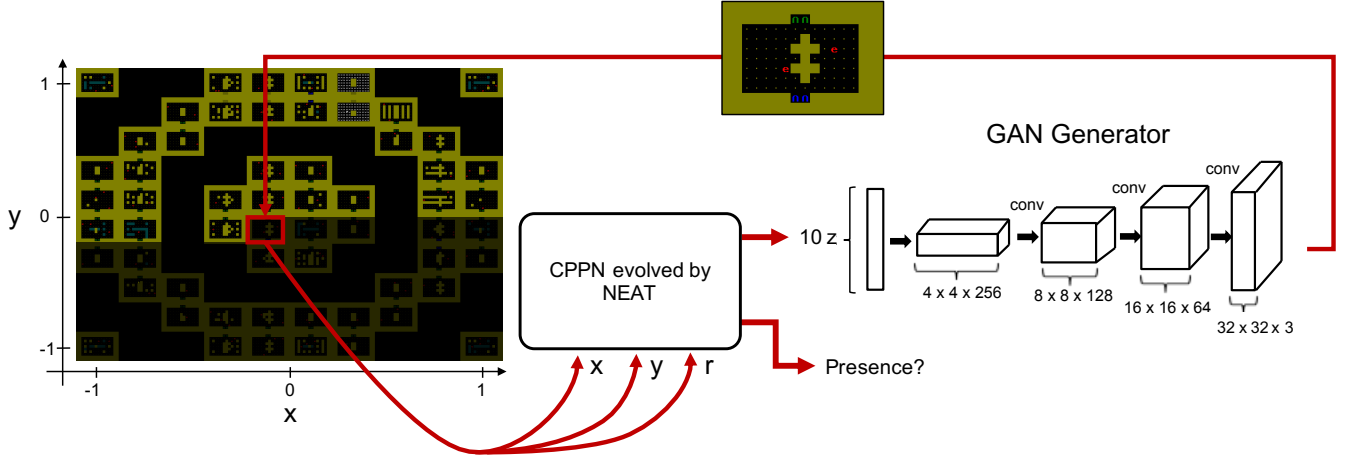[1]Code available at https://github.com/schrum2/GameGAN

Figure 1: CPPN2GAN approach applied to Zelda. The CPPN takes as input the Cartesian coordinates of a segment in a level (x and y) together with its distance from the center r, and for each segment produces a different latent vector z that is then fed into the generator of a GAN pre-trained on existing level content. The CPPN also outputs additional information determining whether the room should be placed, how its doors connect to other rooms, and other miscellaneous information. The approach is able to capture the patterns in the individual level segments, but also create complete maps with global structures such as imperfect radial symmetry in the example here.

CPPN2GAN approach in terms of coverage of level design space, and also does not scale to arbitrary level sizes like CPPN2GAN.

Ultimately, the new CPPN2GAN approach could be relevant not only for game levels, but for other domains requiring large-scale pattern generators, such as texture generation, building architecture, neural architecture search, or computer-aided design in general.

2 RELATED WORK

This paper combines Generative Adversarial Networks (GANs) and Compositional Pattern Producing Networks (CPPNs) into a new form of Latent Variable Evolution (LVE).

2.1 CPPNs

Compositional Pattern Producing Networks (CPPNs [21]) are artificial neural networks (ANNs) with varying activation functions per node. Unlike standard ANNs, they are repeatedly queried across a geometric space of possible inputs. For example, a CPPN can generate a 2D image by taking Cartesian pixel coordinates (x, y) as input and outputting intensity values for each corresponding pixel.

CPPNs typically include a large array of different activation functions that are biased towards specific patterns and regularities. For example, a Gaussian function allows a CPPN output pattern to be symmetric, and including a periodic function such as sine can lead to repeating patterns. Other patterns, such as repetition with variation (e.g. the fingers of the human hand) can be created by combining functions (e.g. sine and Gaussian). CPPNs have been adapted to produce a variety of patterns in domains such as 2D images [20], musical accompaniments [16], 3D objects [7], animations [26], and physical robots [5]. In games, CPPNs have been used to create particle effects for weapons [15] and flowers [19].

CPPNs are traditionally optimized through NeuroEvolution of Augmenting Topologies (NEAT [22]). NEAT starts with a population of simple neural networks: inputs are directly connected to outputs. Throughout evolution, mutations add nodes and connections. NEAT

also allows for efficient crossover between structural components with a shared origin. The benefit of NEAT is that it optimizes both the neural architecture and weights of the network at the same time. More recently, CPPN-inspired neural networks have also been optimized through gradient descent-based approaches [9, 13].

While CPPNs can create patterns with complex regularities, training CPPNs to recreate particular images is difficult [30]. GANs do not share this weakness, which is why we suggest combining them.

2.2 Generative Adversarial Networks

The training process of Generative Adversarial Networks (GANs [11]) can be seen as a two-player adversarial game in which a generator G and a discriminator D are trained at the same time by playing against each other. The discriminator D's job is to classify samples as being generated (by G) or real. The discriminator aims to minimize classification error, but the generator tries to maximize it. Thus, the generator is trained to deceive the discriminator by generating samples that are good enough to be classified as genuine. Training ideally reaches a steady state where G reliably generates realistic examples and D is no more accurate than a coin flip.

At the end of training, the discriminator D is discarded, and the generator G is used to produce new, novel outputs that capture the fundamental properties present in the training examples. The input to G is some fixed-length vector from a latent space. For a properly trained GAN, randomly sampling vectors from this space should produce outputs that could pass as real images or levels. However, to find content with certain properties (such as a specific game difficulty, number of enemies), the latent space needs to be searched, as described next.

2.3 Latent Variable Evolution

The first latent variable evolution (LVE) approach was introduced by Bontrager et al. [3]. In their work a GAN is trained on a set of real fingerprint images and then evolutionary search is used to find

Table 1: Tile types used in generated Mario levels. Symbol characters (*sym*) come from the modified VGLC encoding, and the numeric identity values (*num*) are then mapped to the corresponding values employed by the Mario AI framework for visualization (*vis*). The numeric identity values are expanded into one-hot vectors when input into the discriminator network during GAN training.

Tile type	sym	num	vis
Stone	X	0	
Breakable	x	1	
Empty (passable)	-	2	
Question Block with coin	q	3	
Question Block with power up	Q	4	
Coin	o	5	
Pipe	t	6	
Piranha Plant Pipe	p	7	
Bullet Bill	b	8	
Goomba	g	9	
Green Koopa	k	10	
Red Koopa	r	11	
Spiny	s	12	

latent vectors that match with subjects in the dataset. In another paper, Bontrager et al. [2] present an interactive evolutionary system, in which users can evolve the latent vectors for a GAN trained on different classes of objects (e.g. faces or shoes). Because the GAN is trained on a specific target domain, it becomes a compact and robust genotype-to-phenotype mapping (i.e. most phenotypes resemble valid domain artifacts) and users were able to guide evolution towards images that closely resembled given target images. This paper introduces the first indirectly encoded LVE approach. Instead of searching directly for latent vectors, parameters for CPPNs are sought. These CPPNs can generate a variety of different latent vectors, conditioned on the locations of level segments.

3 VIDEO GAME DOMAINS

The games explored in this paper rely on data from the Video Game Level Corpus (VGLC [24]). Specifically, GAN models for Super Mario Bros. and The Legend of Zelda are trained, though in each case some specialized processing of the data is required.

3.1 Super Mario Bros.

Super Mario Bros. (1985) is a platform game that involves moving left to right while running and jumping. Levels are visualized with the Mario AI framework[2].

The tile-based level representation comes from VGLC, which uses a particular character symbols to represent each possible tile type. The encoding is extended to more accurately reflect the data in the original game (Table 1). For example, VGLC does not distinguish between different enemy types. Besides adding symbols for more enemy types, the representation of *pipes* is adjusted to avoid the broken *pipes* seen in previous work [29]. Instead of using four

Table 2: Tile types used in generated Zelda rooms. The symbol characters (*sym*) come from the VGLC encoding, and the corresponding tile images from the original Legend of Zelda are also shown (*game*). The diversity of VGLC tile types is mapped down to a smaller set of tiles. The numeric values for GAN training are shown (*num*), and the final tiles depict how they appear in the Rogue-like game engine used for visualization (*rogue*).

Tile type	sym	game	num	rogue
Floor	F		0	
Wall	W		1	
Block	B		1	
Door	D		1	
Stair	S		1	
Monster statue	M		1	
Water	P		2	
Walk-able Water	O		2	
Water Block	I		2	

different tile types for a *pipe*, a single tile is used as an indicator for the presence of a *pipe* and extended automatically downward as required. A detailed explanation of all modifications made to the encoding can be found in work by Volz [27, Chap. 4.3.3.2].

3.2 The Legend of Zelda

The Legend of Zelda (1986) is an action-adventure dungeon crawler. The main character, Link, explores several dungeons full of enemies, traps, and puzzles. This game also has a tile-based VGLC description. In this paper, the game is visualized with an ASCII-based Rogue-like game engine used in previous work [12]. The mapping between original game tiles and Rogue-like tiles is in Table 2.

Previous work [12] reduced the large set of tiles inherent to Zelda to a smaller set based on functional requirements. Some Zelda tiles differ in purely aesthetic ways, and others rely on complicated mechanics not implemented in the Rogue-like. The Rogue-like accommodates three tile types: a floor tile which directly corresponds to the Zelda floor tile, an impassable tile that corresponds to all impassable tiles in Zelda, and a water tile that corresponds to an obstacle that enemies can pass, but Link cannot.

Enemies are not represented in the VGLC data because its authors did not include them[3]. VGLC does include information about doors linking rooms, but that information is excluded from the encoding because door placement is handled not by the GAN, but by CPPNs, as described in Section 4.2.

4 APPROACH

The novel approach introduced in this paper employs a two-stage indirect encoding to evolve video game levels. Individual level segments are created by sending latent vectors to a Generative Adversarial Network (GAN) trained on data for a target game. To create whole levels, locations of individual segments are used as input to

a Compositional Pattern Producing Network (CPPN), which outputs latent vectors for each segment. An overview of the complete CPPN2GAN approach below is shown in Fig. 1. For comparison, the Direct2GAN control approach that generates levels from a genome consisting of many separate latent vectors is also described below.

4.1 GAN Training Details

The Mario model used was taken from a publicly available repository associated with previous research [28], but details of its training are repeated here. The Zelda model is new, but is based on another model from a recent paper [12]. Both models are Wasserstein GANs [1] differing only in the size of their latent vector inputs (10 for Zelda, 30 for Mario), and the depth of the final output layer (3 for Zelda, 13 for Mario). Their architecture otherwise matches that shown in Fig. 1. Output depth corresponds to the number of possible tiles for the game. The other output dimensions are 32×32, which is larger than the 2D region needed to render a level segment. During training and generation, the upper left corner of the output region is treated as a generated level, and the rest is ignored.

To encode the levels for training, each tile type is represented by a distinct integer, which is converted to a one-hot encoded vector before being input into the discriminator. The generator also outputs levels in the one-hot encoded format, which is then converted back to a collection of integers. Mario or Zelda levels in this integer-based format can be sent to the Mario AI framework or Rogue-like engine for rendering. The mapping from VGLC tile types and symbols, to GAN training number codes, and finally to Mario AI/Rogue-like tile visualizations is detailed in Tables 1 and 2.

GAN input files for Mario were created by processing all 12 *overworld* level files from VGLC for Super Mario Bros. Each level is 14 tiles high. The GAN expected to always see a rectangular input of the same size, so each input was generated by sliding a 28 (wide) $\times$ 14 (high) window over the level from left to right, one tile at a time. The width of 28 tiles is the width of the screen in Mario.

The GAN input for Zelda was created from the 18 dungeon files in VGLC for The Legend of Zelda, but the actual training samples are the individual rooms in the dungeons, which are 16 (wide) $\times$ 11 (high) tiles in size. Many rooms are repeated within and across dungeons, so only unique rooms were included in the training set (only 38 samples). The training samples are simpler than the raw VGLC rooms because the various tile types are reduced to a set of just three as shown in Table 2. Doors were transformed into walls because door placement is not handled by the GAN, but rather by the CPPN2GAN approach, described next.

4.2 Level Generation: CPPN2GAN

In order to generate a level using CPPNs and GANs, the CPPN is given responsibility for generating latent vector inputs for the GAN as a function of segment position within the larger level.

For Mario, the only input is the x-coordinate of the segment, scaled to $[-1, 1]$. So, for a level of three segments, the CPPN inputs would be -1, 0, and 1. The output of the CPPN is an entire latent vector (30 variables for Mario). Each latent vector is fed to the GAN to generate the segment at that position in the level.

Zelda's 2D arrangement of rooms is more complicated. For the overall dungeon shape to be interesting, some rooms in the 2D grid

need to be missing. Also, dungeons are typically more interesting if they are maze-like, so simply connecting all adjacent rooms would be boring. How maze-like a dungeon is also depends on its start and end points. These additional issues are global design issues, and therefore are handled by the CPPN, which defines global patterns, rather than the GAN, which generates individual rooms.

Thus, CPPNs for Zelda generate latent vector inputs and additional values that determine the layout and connectivity of the rooms. Zelda CPPNs take inputs of x and y coordinates scaled to $[-1, 1]$. A radial distance input is also included to encourage radial patterns, which is common in CPPNs [20]. For each set of CPPN inputs, the output is a latent vector (size 10 for Zelda) along with six additional numbers: room presence, right door presence, down door presence, right door type, down door type, and start/end preference.

Room presence determines the presence/absence of a room based on whether the number is positive. Similarly, if a room is present and has a neighboring room in the given direction, then positive right/down door presence values place a door in the wall heading right/down. Whenever a door is placed, a door is also placed in the opposite direction within the connecting room, which is why top/left door outputs are not needed. For variety, the right/down door type determines the types of doors, based on different number ranges for each door type: $[-1, 0]$ for plain, $(0, 0.33]$ for soft-locked, $(0.33, 0.66]$ for bomb-able passage, and $(0.66, 1.0]$ for locked. Soft-locked doors only open when all enemies in the room are killed, bomb-able passages are secret walls that can be bombed to create a door, and locked doors need a key. Enough keys to pair with all locked doors are placed at random locations in rooms of the dungeon. However, to assure that the genotype to phenotype mapping is deterministic, the pseudo-random generator responsible for placing keys is initialized using the bit representation of the corresponding right or down door type output as a seed.

The final output for start/end preference determines which rooms are the start/end rooms of the dungeon. Across all rooms in the dungeon, the one whose start/end preference is smallest is the player's starting room, and the one with the largest output is the final goal room, designated by the presence of a Triforce item (triangle).

This new approach to generating complete levels is compared with the control approach described next.

4.3 Level Generation: Direct2GAN

To have a meaningful comparison with the CPPN2GAN approach, an approach that directly evolves genomes consisting of multiple latent vectors is needed. The Direct2GAN method evolves levels consisting of S segments for a GAN expecting latent inputs of size Z by evolving real-valued genome vectors of length $S \times Z$. Each genome is chopped into individual GAN inputs at level generation.

This approach requires a convention as to how different segments are combined into one level. For Mario's linear levels, adjacent GAN inputs from the combined vector correspond to adjacent segments in the generated level. The combined vector is simply processed from left to right to produce segments from left to right.

To generate 2D Zelda dungeons, individual segments of the linear genome are mapped to a 2D grid in row-major order: processing genome from left to right generates top row from left to right, then moves to next row down and so on. For fair comparison with

CPPN2GAN, each portion of a genome corresponding to a single room contains not only the latent vector inputs, but the six additional numbers for controlling global structure and connectivity: room presence, right door presence, down door presence, right door type, down door type, and start/end preference. Therefore, a $M \times N$ room grid requires genomes of length $M \times N \times (Z + 6)$.

Such massive genomes induce large search spaces that are difficult to search, as demonstrated by the experiments described next.

5 EXPERIMENTS

Demonstrating the expressive range of new game level encodings is important. Therefore, the Quality Diversity algorithm MAP-Elites [17], which divides the search space into phenotypically distinct bins, is used for evolving both CPPNs and real-valued vectors.

5.1 MAP Elites

Instead of only optimizing towards an objective, as in standard evolutionary algorithms, MAP Elites (Multi-dimensional Archive of Phenotypic Elites [17]) collects a diversity of quality artefacts that differ along a number N of predefined dimensions. MAP-Elites discretizes the space of produced artefacts into different bins and, given some objective, keeps track of the highest performing individual for each bin in the N-dimensional behavior space.

Our implementation starts by generating an initial population of 100 random individuals that are placed in bins based on their attributes. Each bin only holds one individual, so individuals with higher fitness replace less fit individuals. Once the initial population is generated, solutions are randomly sampled uniformly from the bins and undergo crossover and/or mutation to generate new individuals. These newly created individuals also replace less fit individuals as appropriate, or end up occupying new bins, filling out the range of possible designs. Performance is measured both in terms of achieved fitness and the number of bins that are filled as more individuals are generated. Our experiments generate 50,000 individuals per run after the initial population is generated.

To support a range of meaningful variation within each domain, each game uses its own distinct binning scheme and fitness measure.

5.2 Dimensions of Variation Within Levels

In Mario, the bin dimensions are based on measurements of three quantities: decoration frequency, space coverage, and leniency. These measures were inspired by a study on evaluation measures for platformer games [23]. Each measure expresses different characteristics of a level:

- decoration frequency: Percentage of *non-standard* tiles[4]
- space coverage: Percentage of tiles that Mario can stand on[5]
- leniency: Average of leniency values[6] across all tiles. Enemies and gaps have negative values, power-ups positive values.

Each score highlights different aspects of a level. All measures focus on visual characteristics, but also relate to how a player can navigate through a level. These scores were calculated for individual segments (10 per level) and then summed across the segments.

---

[4]breakable tiles, question blocks, pipes, all enemies
[5]solid and breakable tiles, question blocks, pipes and bullet bills
[6]1: question blocks; -0.5: pipes, bullet bills, gaps in ground; -1: moving enemies; 0: remaining

Preliminary experiments were conducted to uncover reasonable ranges for binning. Multiplying decoration frequency and space coverage by 3.0 scales them roughly to the range $[0, 1]$, which is evenly split into 10 bins. Only leniency has both negative and positive values. Multiplying the raw score by 5.0 scales it roughly to the range $[-0.5, 0.5]$, which is evenly divided into 5 negative bins and 5 positive bins. Negative bins correspond to greater challenges, and positive bins correspond to easier levels. In each dimension, rare out-of-range values were assigned to the nearest bin.

The fitness is the length of the shortest path to beat the level. The objective is to maximize the path length, which favors levels that require jumps - the main mechanic of the game. If no path can be found the level is deemed unsolvable and receives a fitness of 0.

To determine the path, A* search is performed on the tile-based representation of the level. To limit computational costs, the game physics are simplified as follows. First, Mario always occupies a single tile (small state), and can either move left, move right, or jump. Once a jump starts, Mario moves up one tile per action for the next four actions (unless there is an obstruction), and then falls down until landing on an impassable tile. Mario can still make left/right movements while airborne, but can only initiate a jump if on solid ground. Enemies are just impassable obstacles, and Mario dies if he falls in a pit. The heuristic used favors states further to the right, because the right edge of the screen is the goal.

Variation across Zelda levels is based on water tile percentage, wall tile percentage, and the number of reachable rooms. A room is reachable if it is the start room, or if a door connects it to any reachable room. This definition makes it computationally cheap to determine if a room is *reachable*, but ignores that single rooms can be impassable[7]. Water and wall tile percentages are calculated only with respect to reachable rooms, and only for the 12×7 floor regions of rooms (surrounding walls ignored). Bins for these dimensions are divided into 10% ranges, creating 10 bins per dimension. However, some of these bins are impossible to fill, because the sum of water and wall percentages must be less than 100% to fit in the room. Floor tiles occupy additional space. For the number of reachable rooms, there is a bin for each possible number out of 100. The maximum number is 100 because dungeons are generated in a 10×10 grid.

The fitness for Zelda dungeons is the percentage of reachable rooms traversed by the shortest solution path from start to goal. The objective is to maximize the number of rooms visited, as exploring is one of the main mechanics of the game. Again, if no path can be found, the dungeon is deemed unsolvable and receives 0 fitness.

A* is again used to determine the path, now using Manhattan distance to the goal as a heuristic. We take locked doors and keys into account to only generate paths with rooms that are *actually* reachable. Since the inclusion of keys makes the state space very large, there is a computation budget of 100,000 states.

5.3 CPPN Evolution Details

Levels for both games are evolved using a variant of NEAT [22], as described in Section 2.1. The specific implementation is MM-NEAT[8]. Because CPPNs are being evolved, every neuron in each network can have a different activation function from the following

---

[7]For example because a needed key is inaccessible
[8]https://github.com/schrum2/MM-NEAT
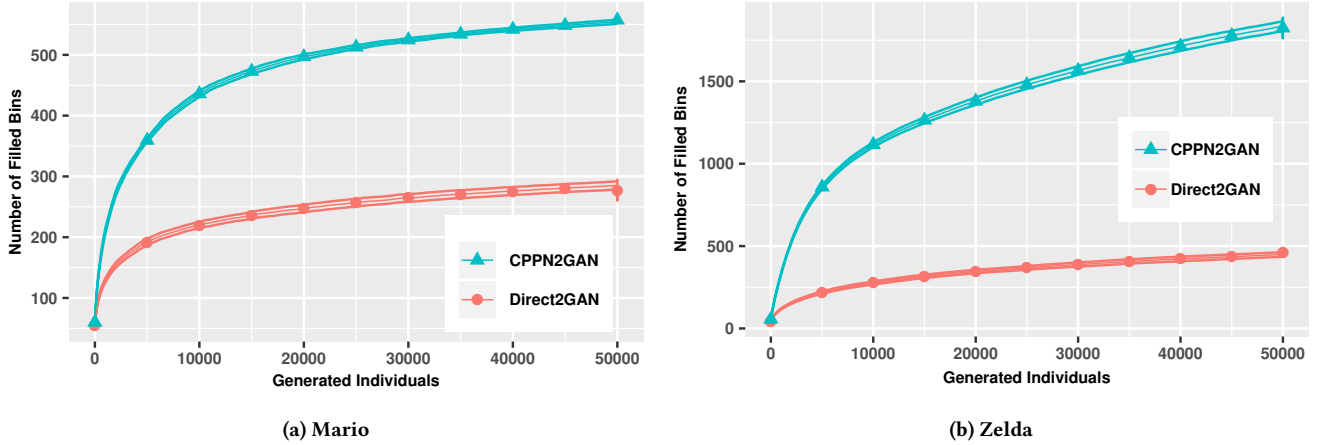
(a) Mario

(b) Zelda

Figure 2: Average Number of Filled Bins. Depicts the average number of bins filled across 30 runs as more levels are generated for both (a) Mario, and (b) Zelda. The 95% confidence intervals are also shown, and are very narrow, indicating high consistency in performance. CPPN2GAN fills significantly more bins than Direct2GAN in both domains for nearly the entirety of evolution.

list: sawtooth wave, linear piecewise, id, square wave, cosine sine, sigmoid, Gaussian, triangle wave, and absolute value.

Whenever a new network is generated, is has a 50% chance of being the offspring of two parents rather than a clone. The resulting network then has a 20% chance of having a new node spliced in, 40% chance of creating a new link, and a 30% chance of randomly replacing one neuron's activation function. There is a per-link perturbation rate of 5%.

5.4 Real-Valued Evolution

For Direct2GAN, real-valued vectors are initialized with random values in the range $[-1, 1]$. When offspring are produced, there is a 50% chance of single-point crossover. Otherwise, the offspring is a clone of one parent. Either way, each real number in the vector then has an independent 30% chance of polynomial mutation [8].

6 RESULTS

In terms of the number of MAP Elites bins filled during search, CPPN2GAN quickly surpasses Direct2GAN by a statistically significant amount ($p < 0.05$) in both Mario and Zelda (Fig. 2). The differences are large, and confidence intervals are narrow.

Final heatmaps from individual runs demonstrate which regions of design space each method occupies. Fig. 3 shows final results in Mario for typical runs of Direct2GAN and CPPN2GAN. Direct2GAN seems to have trouble discovering levels with a positive leniency. This is likely because the only way to reach these high leniency values is by adding lots of question blocks and few enemies, which limits the number of suitable latent vectors. Here CPPN2GAN benefits from its encoding, which makes it easier to re-use discovered good solutions. Additionally, even in bins with negative leniency, CPPN2GAN fills in a wider range of decoration frequency and space coverage values, and with higher fitness. Interestingly, both methods have trouble discovering levels with both low space coverage and high decoration frequency. This is because there are several decorative tiles that Mario can stand on (pipes, bullet bills, question

blocks, breakable tiles) which thus automatically increase space coverage as well. Example evolved levels are in Fig. 4.

Example Zelda heatmaps are in Fig. 5. They reveal that Direct2GAN has trouble creating dungeons with many reachable rooms, whereas CPPN2GAN successfully discovers dungeons with every possible number of reachable rooms. Even for smaller dungeons, CPPN2GAN fills out the range of possible water and wall percentages more fully, and with higher fitness. Example evolved dungeons are in Fig. 6. Direct2GAN has trouble connecting scattered rooms, whereas CPPN2GAN, as expected, lays out rooms in a cohesive pattern with more uniformly structured connectivity.

These experiments demonstrate that CPPN2GAN is vastly superior to Direct2GAN in terms of how much of the design space is covered for both Mario and Zelda. Further, if specific rare level characteristics are desired (e.g. Mario levels with high leniency), CPPNs are more likely to discover suitable levels due to indirect encoding. However, if diversity within the generated content is desired, Direct2GAN is the better approach. While Mario levels generated with CPPN2GAN do switch between patterns as shown in Fig. 4b, Direct2GAN creates more variation (Fig. 4a). This of course comes with the cost of losing global level structure.

Similarly, CPPN2GAN is better able to discover Zelda dungeons with specific properties (such as water and wall percentage), but CPPN2GAN is better at producing variation in Zelda than in Mario. Rooms are still frequently repeated, but not as extensively. Confined geographic regions in each dungeon feature particular tile configurations more prominently, and smoothly transition from one region to another in an organic fashion. This result is especially interesting due its potential correlation with the in-game narrative. For instance, the dungeon in Fig. 6b starts in a region of water-filled maze-like rooms, leads the player through rooms with a more open floor plan, then proceeds through a region of different water obstacles before a wide open set of rooms funnels the player to the Triforce. The dungeon in Fig. 6c is interesting because the solution path has several twists and turns, despite all rooms being reachable. This path results from how the CPPN used the geometry of the grid
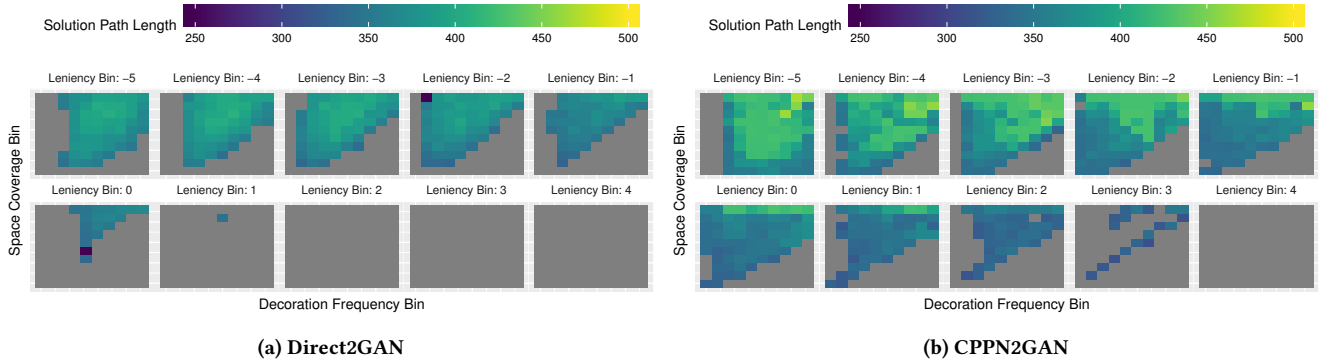
(a) Direct2GAN

(b) CPPN2GAN

Figure 3: Heatmaps of Typical Final Archives for Mario Level Evolution. Final MAP-Elites archives are depicted for one run each of (a) Direct2GAN, and (b) CPPN2GAN. CPPN2GAN fills bins with positive leniency that Direct2GAN cannot fill. Both approaches fail to create levels with high decoration and low space coverage, since such features contradict each other. Even in negative leniency regions, CPPN2GAN fills more bins and with higher fitness.
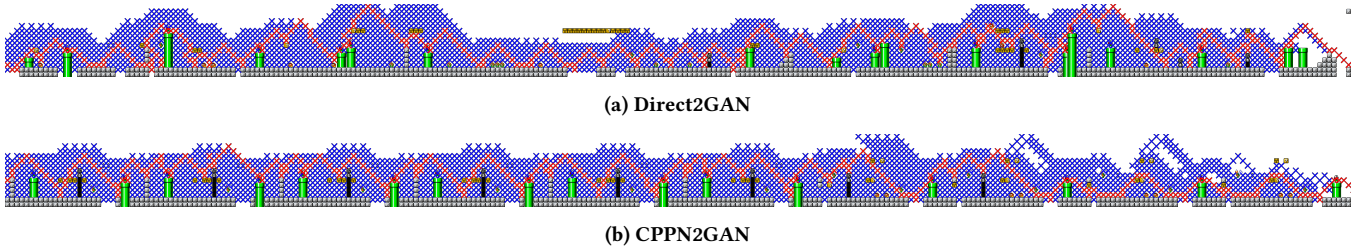

(a) Direct2GAN

(b) CPPN2GAN

Figure 4: Example Evolved Mario Levels. Each level is shown with blue X marks at positions checked by A* search, and a trail of red X marks along the solution path. Both (a) Direct2GAN and (b) CPPN2GAN levels are shown. These levels are in the same bin, but from different runs. The CPPN2GAN level begins with one repeating pattern of segments, but then switches to another in the latter half.

to restrict room connectivity. However, this dungeon also has an interesting diversity of room layouts. It is possible that the selection pressure to develop interesting room connectivity patterns inserted so many diverse activation functions into the the CPPNs that more diverse room patterns emerged as a side-effect. The Direct2GAN dungeon (Fig. 6a) also features diverse room patterns, but they are not arranged in a cohesive fashion, and this approach is also more susceptible to creating isolated, disconnected rooms.

7   CONCLUSIONS AND FUTURE WORK

Generative Adversarial Networks (GANs) have shown impressive results as generators for high-quality images. However, combining multiple GAN-generated patterns into a cohesive whole, which is especially relevant for level generation in games, was so far an unexplored area of research. This paper presents the first method that can create large-scale game levels through the combination of a Compositional Pattern Producing Network and a GAN. One of the main insights is that there is a functional relationship between the latent vectors of different game segments, which the CPPN is able to exploit. Additionally, in comparison to a direct representation of multiple latent vectors, the introduced CPPN2GAN approach was able to generate a larger variety of different complete game levels.

However, CPPN2GAN results often contained repeated segments. If this is undesirable for different applications, future experiments could use measures for solution variety that also consider local

diversity. Such measures might do a better job of encouraging repetition *with* variation instead of pure repetition.

Beyond our proposed approach, there are other interesting ways to combine the two techniques in this paper. For example, CPPNs could be used to generate global structure only, while modules are evolved independently to fulfill local objectives. For Zelda, this would mean that the global dungeon structure (room presence and their connections) is generated via CPPN, but the separate rooms are determined by evolved real vectors. Such a combination would create interesting global layouts, but allow for arbitrary, chaotic variation when going from room to room.

The two approaches could also be combined sequentially. First, CPPN2GAN would introduce some initial global structure and impose some regular patterns in the level. Afterwards, Direct2GAN could take over to fine-tune the results and/or introduce more local variety. This idea is similar to HybrID [6].

However, these extensions lose the ability to scale to arbitrary sizes; a major benefit of CPPNs. In fact, the CPPN2GAN encoding could enable levels in which components are lazily generated as needed. This would be especially useful for exploration games, as new modules (e.g. dungeon rooms) can be served by CPPN2GAN whenever new areas of the map are discovered by the player.

Besides different combination approaches, modifications to the training process are also possible. Here, the GAN was pre-trained and only the CPPNs were evolved. However, it should be possible to make use of a discriminator to also decide whether global patterns
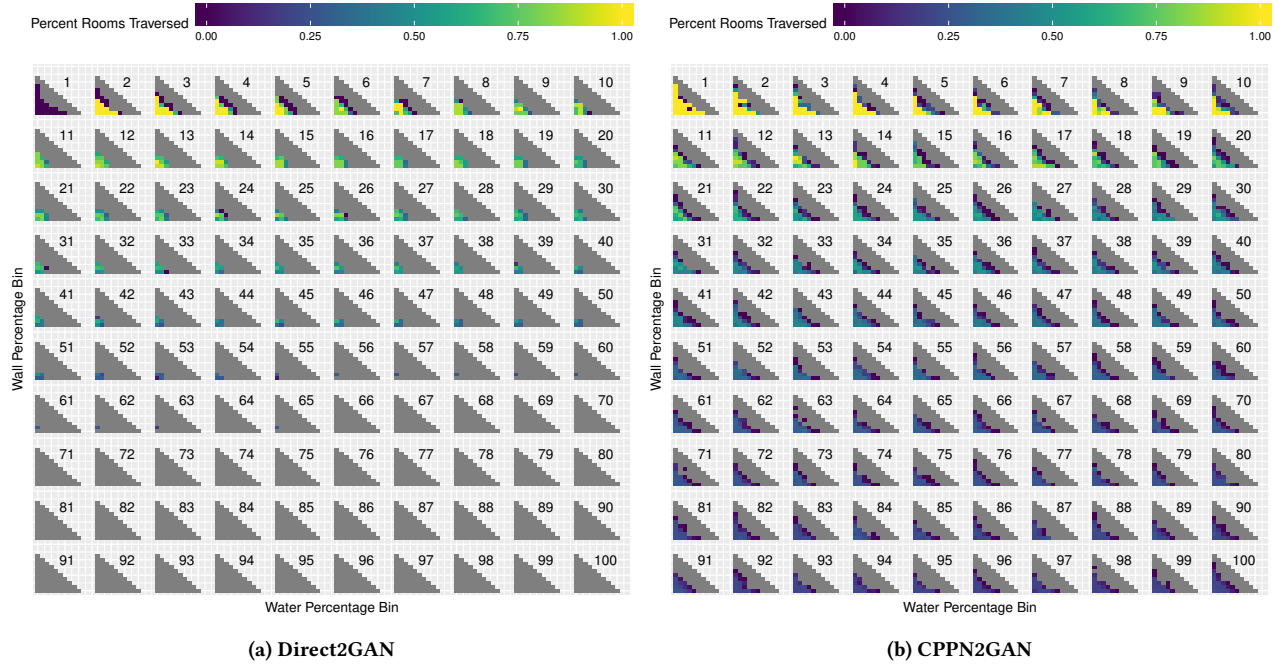
(a) Direct2GAN

(b) CPPN2GAN

Figure 5: Heatmaps of Typical Final Archives for Zelda Dungeon Evolution. Final MAP-Elites archives are depicted for one run each of (a) Direct2GAN, and (b) CPPN2GAN. Each square corresponds to dungeons having a different number of reachable rooms, shown in the upper-right corners. Within each square are 10 columns for the percentage of room tiles that are water, partitioned into groups of 10%. The rows of each square indicate the percentage of wall tiles in a similar way. The color intensity is the fitness of the individual in the bin: the percent of reachable rooms traversed from start to goal. Note that the upper-right corners of each square are impossible to fill, because the sum of water and wall percentage must be less than 100%. CPPN2GAN has multiple representatives with every possible number of reachable rooms, but Direct2GAN struggles to create dungeons with larger numbers of reachable rooms.
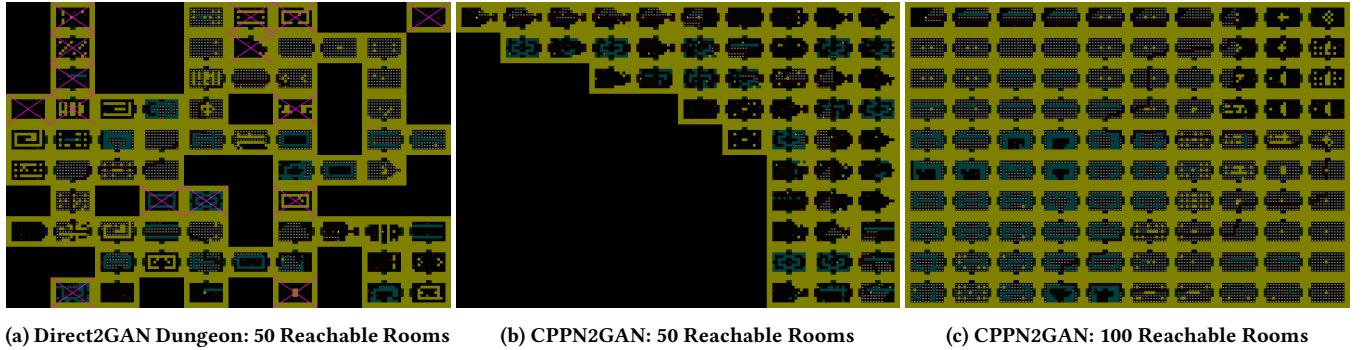

(a) Direct2GAN Dungeon: 50 Reachable Rooms   (b) CPPN2GAN: 50 Reachable Rooms   (c) CPPN2GAN: 100 Reachable Rooms

Figure 6: Example Evolved Dungeons. The (a) Direct2GAN and (b) CPPN2GAN dungeons with 50 reachable rooms each have the same fitness (38% of reachable rooms traversed in solution path). There is also a (c) CPPN2GAN dungeon with 100 reachable rooms and a comparable fitness of 34% reachable rooms. Direct2GAN could not produce dungeons with so many reachable rooms. The Direct2GAN dungeon is more sprawling, and has several rooms that are not reachable (magenta X). Both CPPN2GAN dungeons are more cohesive, and themes can be noticed in different regions of the dungeons. Larger versions of these figures are included in supplementary material.

(such as Zelda room structure) are similar to the original levels. This would mean adversarial training of the complete CPPN2GAN network against a discriminator. Training could be end-to-end or by alternating between the CPPN and the generator. The resulting samples should be able to reproduce both global *and* local patterns in complete game levels rather than just in individual segments.

ACKNOWLEDGMENTS

REFERENCES

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein Generative Adversarial Networks. In *International Conference on Machine Learning*.

[2] Philip Bontrager, Wending Lin, Julian Togelius, and Sebastian Risi. 2018. Deep Interactive Evolution. *European Conference on the Applications of Evolutionary Computation (EvoApplications)*.

[3] Philip Bontrager, Julian Togelius, and Nasir Memon. 2017. DeepMasterPrint: Generating Fingerprints for Presentation Attacks. *arXiv:1705.07386* (2017).

[4] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2018. Large Scale GAN Training For High Fidelity Natural Image Synthesis. *arXiv:1809.11096* (2018).

[5] Daniel Cellucci, Robert MacCurdy, Hod Lipson, and Sebastian Risi. 2017. 1D Printing of Recyclable Robots. *IEEE Robotics and Automation Letters* 2, 4 (2017).

[6] Jeff Clune, Benjamin E. Beckmann, Robert T. Pennock, and Charles Ofria. 2009. HybrID: A Hybridization of Indirect and Direct Encodings for Evolutionary Computation. In *European Conference on Artificial Life*. 134–141.

[7] Jeff Clune and Hod Lipson. 2011. Evolving Three-dimensional Objects with a Generative Encoding Inspired by Developmental Biology. In *European Conference on Artificial Life*. 141–148.

[8] Kalyanmoy Deb and Ram Bhushan Agrawal. 1995. Simulated Binary Crossover For Continuous Search Space. *Complex Systems* 9, 2 (1995), 115–148.

[9] Chrisantha Fernando, Dylan Banarse, Malcolm Reynolds, Frederic Besse, David Pfau, Max Jaderberg, Marc Lanctot, and Daan Wierstra. 2016. Convolution by Evolution: Differentiable Pattern Producing Networks. In *Genetic and Evolutionary Computation Conference*. 109–116.

[10] Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. 2019. Searching the Latent Space of a Generative Adversarial Network to Generate DOOM Levels. In *Conference on Games*. IEEE, 1–8.

[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*. 2672–2680.

[12] Jake Gutierrez and Jacob Schrum. 2020. Generative Adversarial Network Rooms in Generative Graph Grammar Dungeons for The Legend of Zelda. In *Congress on Evolutionary Computation*. IEEE.

[13] David Ha, Andrew Dai, and Quoc V. Le. 2017. HyperNetworks. In *International Conference on Learning Representations*.

[14] N. Hansen and A. Ostermeier. 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* 9, 2 (2001), 159–195.

[15] Erin Jonathan Hastings, Ratan K Guha, and Kenneth O Stanley. 2009. Automatic Content Generation in the Galactic Arms Race Video Game. *IEEE Transactions on Computational Intelligence and AI in Games* 1, 4 (2009), 245–263.

[16] Amy K Hoover, Paul A Szerlip, Marie E Norton, Trevor A Brindle, Zachary Merritt, and Kenneth O Stanley. 2012. Generating a Complete Multipart Musical Composition from a Single Monophonic Melody with Functional Scaffolding. In *International Conference on Computational Creativity*. 111–118.

[17] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating Search Spaces by Mapping Elites. *arXiv:1504.04909* (2015).

[18] Kyungjin Park, Bradford W. Mott, Wookhee Min, Kristy Elizabeth Boyer, Eric N. Wiebe, and James C. Lester. 2019. Generating Educational Game Levels with Multistep Deep Convolutional Generative Adversarial Networks. In *Conference on Games*. IEEE, 1–8.

[19] Sebastian Risi, Joel Lehman, David B D'Ambrosio, Ryan Hall, and Kenneth O Stanley. 2012. Combining Search-Based Procedural Content Generation and Social Gaming in the Petalz Video Game. In *Artificial Intelligence and Interactive Digital Entertainment*.

[20] Jimmy Secretan, Nicholas Beato, David B. D'Ambrosio, Adelein Rodriguez, Adam Campbell, Jeremiah T. Folsom-Kovarik, and Kenneth O. Stanley. 2011. Picbreeder: A Case Study in Collaborative Evolutionary Exploration of Design Space. *Evolutionary Computation* 19, 3 (2011), 373–403.

[21] Kenneth O. Stanley. 2007. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines* 8, 2 (2007), 131–162.

[22] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* (2002).

[23] Adam Summerville, Julian R. H. Mariño, Sam Snodgrass, Santiago Ontañón, and Levi H. S. Lelis. 2017. Understanding Mario: An Evaluation of Design Metrics for Platformers. In *Foundations of Digital Games*. ACM. https://doi.org/10.1145/3102071.3102080

[24] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontañón. 2016. The VGLC: The Video Game Level Corpus. In *Procedural Content Generation in Games*. ACM.

[25] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2019. Bootstrapping Conditional GANs for Video Game Level Generation. *arXiv:1910.01603* (2019).

[26] Isabel Tweraser, Lauren E Gillespie, and Jacob Schrum. 2018. Querying Across Time to Interactively Evolve Animations. In *Genetic and Evolutionary Computation Conference*. ACM.

[27] Vanessa Volz. 2019. *Uncertainty Handling in Surrogate Assisted Optimisation of Games*. Ph.D. Dissertation. TU Dortmund University, Germany.

[28] Vanessa Volz, Boris Naujoks, Pascal Kerschke, and Tea Tušar. 2019. Single- and Multi-Objective Game-Benchmark for Evolutionary Algorithms. In *Genetic and Evolutionary Computation Conference*. ACM, 647–655.

[29] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam M. 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*.

[30] Brian G Woolley and Kenneth O Stanley. 2011. On the Deleterious Effects of A Priori Objectives on Evolution and Representation. In *Genetic and Evolutionary Computation Conference*. ACM, 957–964.
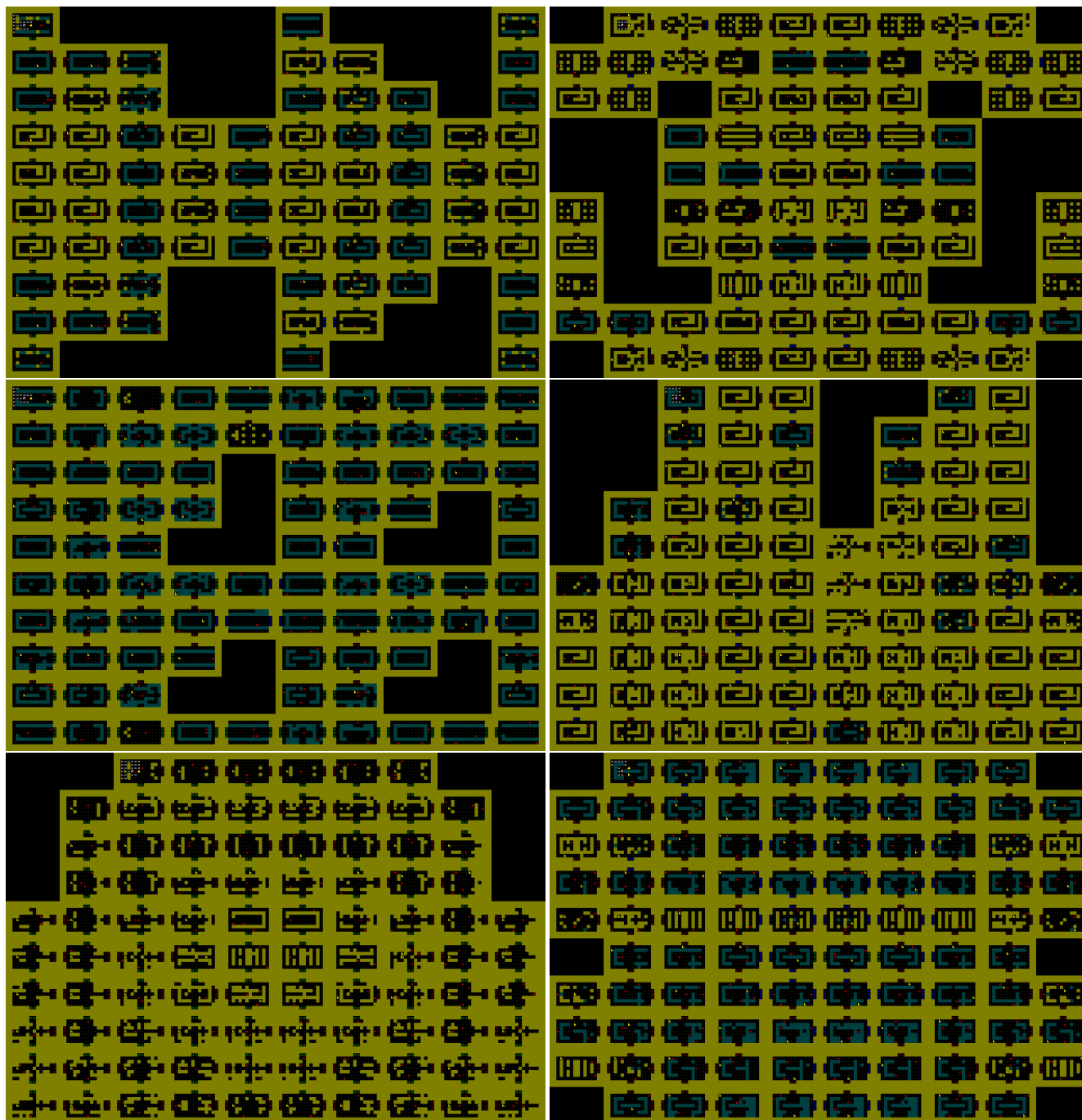
Figure 7: A selection of example Zelda levels created by CPPN2GAN. The maps show varying degrees of symmetry, repetition, and repetition with variation.
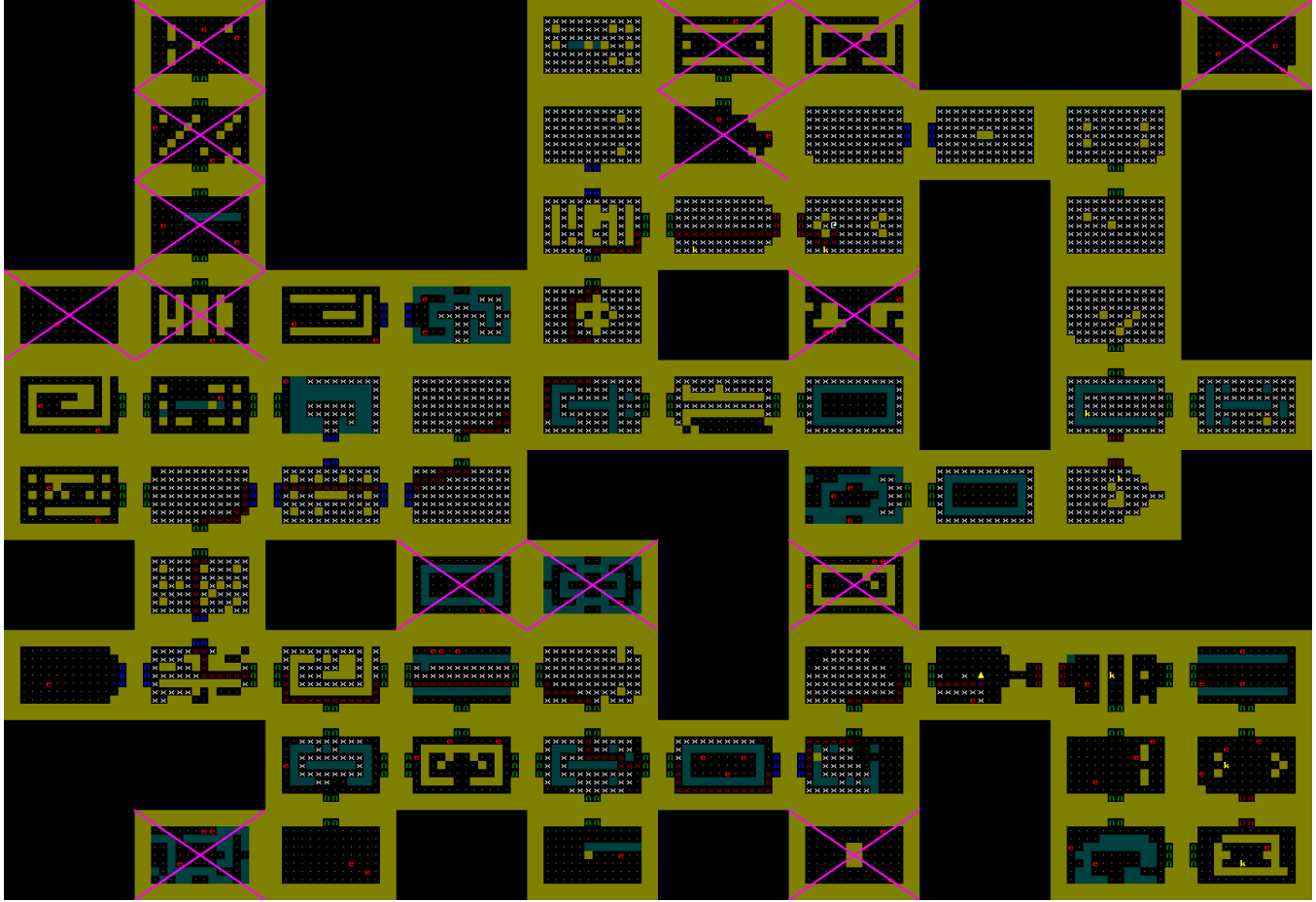
Figure 8: Direct2GAN Dungeon: 50 Reachable Rooms (large version of Fig. 6a). The @ symbol is the start point, and the triangle is the goal. The red X marks from the start to the goal mark the solution path, and the white X marks correspond to locations checked by A* search in order to find the solution. The Direct2GAN dungeon is more sprawling, and has several rooms that are not reachable (magenta X). CPPN2GAN dungeons are more cohesive, and themes can be noticed in different regions of the dungeons.
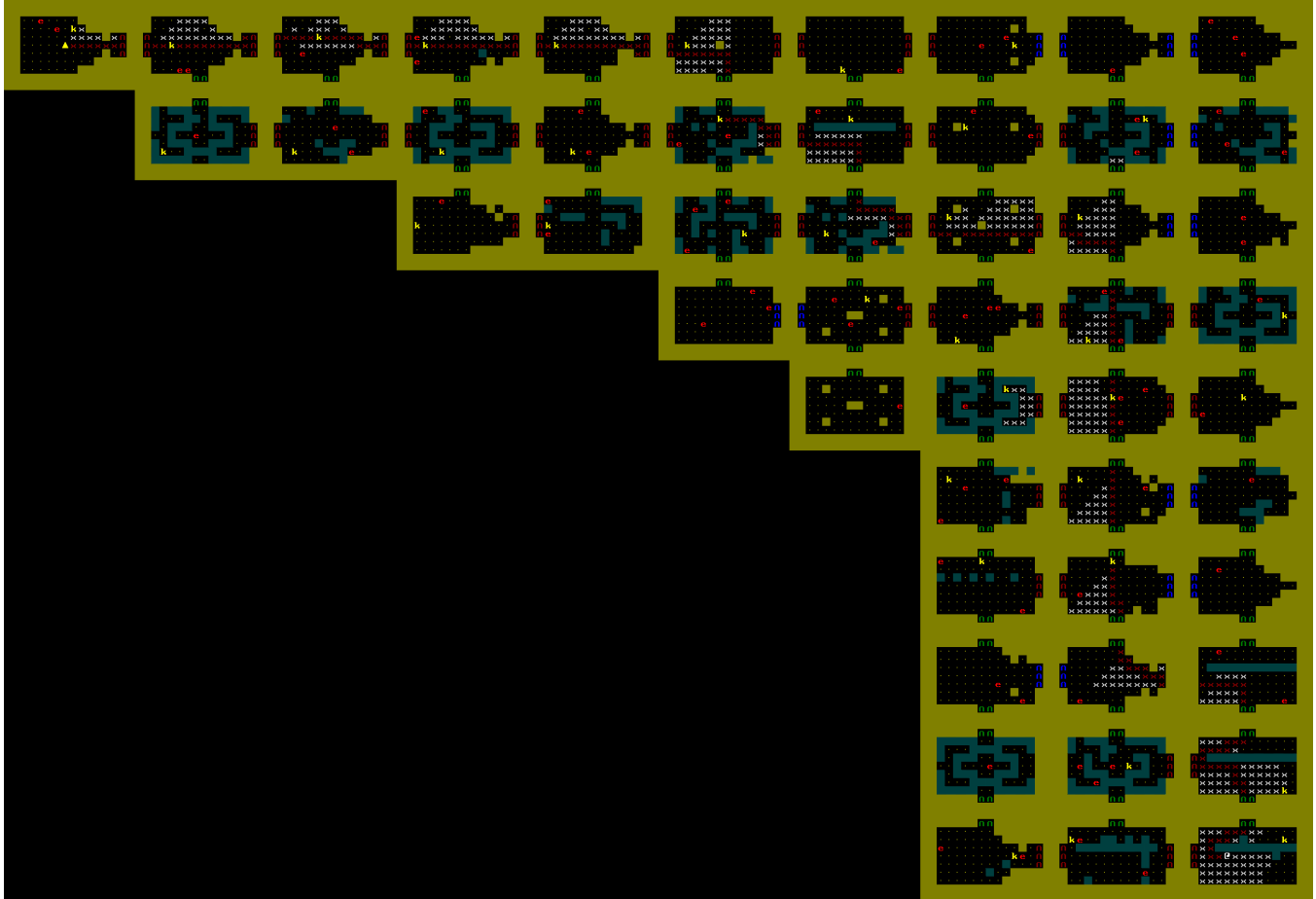
Figure 9: CPPN2GAN: 50 Reachable Rooms (large version of Fig. 6b). The @ symbol is the start point, and the triangle is the goal. The red X marks from the start to the goal mark the solution path, and the white X marks correspond to locations checked by A* search in order to find the solution. CPPN2GAN dungeons are more cohesive, and themes can be noticed in different regions of the dungeons.
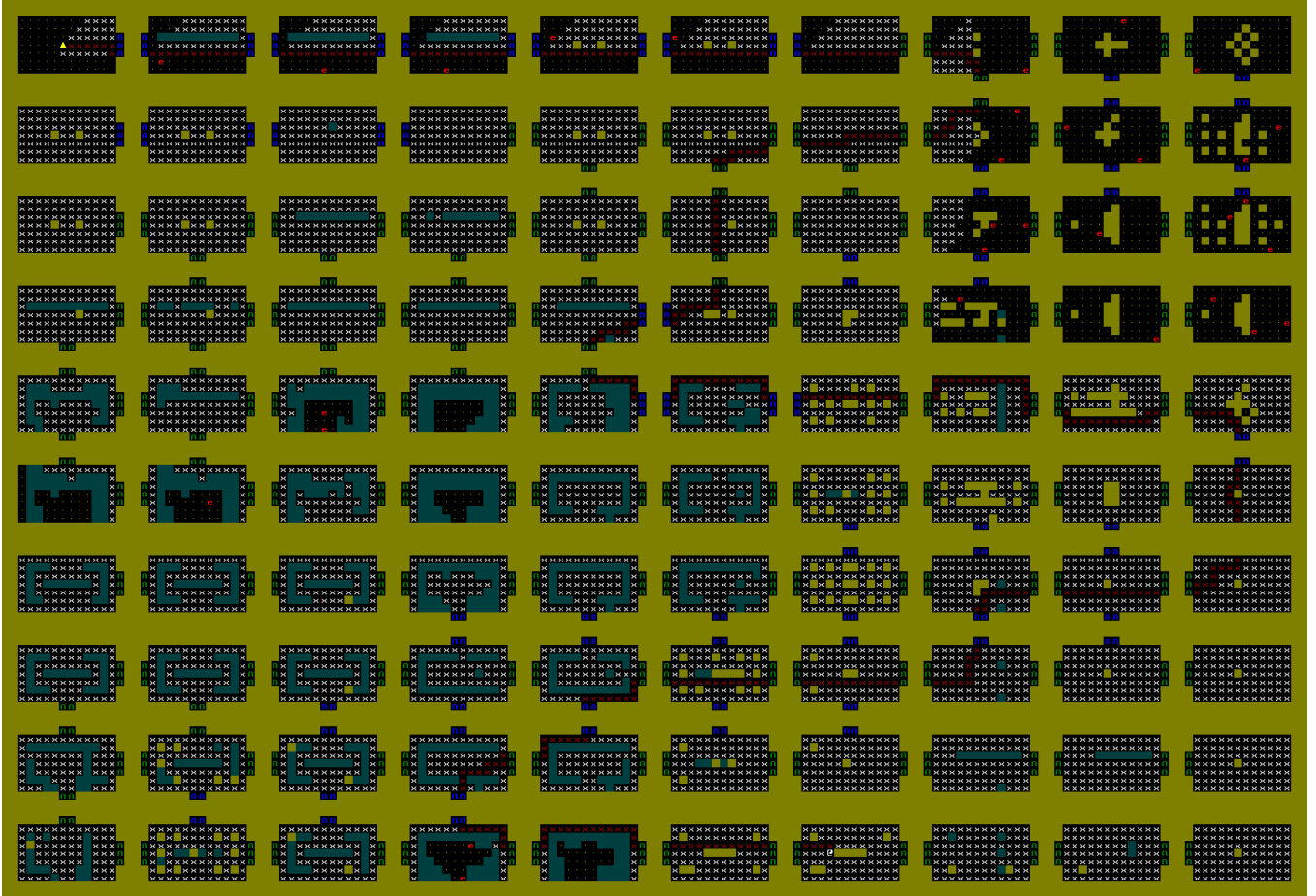
Figure 10: CPPN2GAN: 100 Reachable Rooms (large version of Fig. 6c). The @ symbol is the start point, and the triangle is the goal. The red X marks from the start to the goal mark the solution path, and the white X marks correspond to locations checked by A* search in order to find the solution. CPPN2GAN dungeons are more cohesive, and themes can be noticed in different regions of the dungeons.