Procedural Content Generation using Neuroevolution and Novelty Search for Diverse Video Game Levels

Michael Beukman
University of the Witwatersrand
School of Computer Science and Applied Mathematics
Johannesburg, South Africa
michael.beukman1@students.wits.ac.za

Christopher W Cleghorn
University of the Witwatersrand
School of Computer Science and Applied Mathematics
Johannesburg, South Africa
christopher.cleghorn@wits.ac.za

Steven James
University of the Witwatersrand
School of Computer Science and Applied Mathematics
Johannesburg, South Africa
steven.james@wits.ac.za

ABSTRACT

Procedurally generated video game content has the potential to drastically reduce the content creation budget of game developers and large studios. However, adoption is hindered by limitations such as slow generation, as well as low quality and diversity of content. We introduce an evolutionary search-based approach for evolving level generators using novelty search to procedurally generate diverse levels in real time, without requiring training data or detailed domain-specific knowledge. We test our method on two domains, and our results show an order of magnitude speedup in generation time compared to existing methods while obtaining comparable metric scores. We further demonstrate the ability to generalise to arbitrary-sized levels without retraining.

CCS CONCEPTS

• **Computing methodologies → Genetic algorithms**.

KEYWORDS

Neuroevolution, Novelty Search, Procedural Content Generation

ACM Reference Format:
Michael Beukman, Christopher W Cleghorn, and Steven James. 2022. Procedural Content Generation using Neuroevolution and Novelty Search for Diverse Video Game Levels. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3512290.3528701

1 INTRODUCTION

Video games are a massive industry, with 227 million reported video game players in the United States as of 2021 [9]. Some of the main goals of video games are to keep players entertained, engaged, and challenged [38]. This can be achieved by populating the game with a large amount of unique and interesting content. Nonetheless, most commercial video games still rely on human designers and developers to create this content [17]. However, this is costly, and due to tight deadlines there is always a finite amount of content that players can quickly exhaust. This concern can be addressed through procedural content generation (PCG), where game content is algorithmically created [60].

There are many notable games in which content and levels are procedurally generated, such as *Rogue*, where the player controls a character that traverses dungeons while fighting enemies. Such an approach can be used to generate a near-infinite number of levels—designers need only specify the mechanics of the game and the generation method. This enables game developers to create engaging and fun games at a fraction of the cost compared to traditional, manual development [17].

Outside of game development, PCG can also be leveraged to train machine learning agents.

For example, Justesen et al. [21] train reinforcement learning (RL) agents on procedurally generated levels to improve generalisation to unseen human generated levels, while Cobbe et al. [5] demonstrate that RL agents can overfit on surprisingly large training sets, and use a large number of procedurally generated levels to overcome this.

There are many different approaches to developing PCG algorithms. These include ad hoc algorithms (used in *Rogue*), formal languages [34], evolutionary search-based methods [57, 60], exhaustive search [52], and more recent machine learning approaches [30, 45, 53]. Different methods also have different goals, such as providing a consistent quality of levels, or quickly generating playable [23], diverse [29] or configurable levels [10].

Despite the variety of approaches, there are gaps in the current literature. For example, some approaches generate diverse, playable levels, but the generation process is slow [29, 40]. Some methods can generate levels relatively quickly, but they require existing training data [41, 61] or game-specific reward engineering [23], while others are limited by the lack of diverse content [14, 53]. The main limitation is that no one method can quickly generate diverse levels without the need for training data and game-specific knowledge.

We address this gap by training a neural network that can quickly be queried to generate levels. We avoid the need for training data by using NeuroEvolution of Augmenting Topologies [50] to evolve this network. To obtain diverse levels, we explicitly reward diversity in the evolutionary process by using novelty search [25]. This method assigns fitnesses based on how far an individual is from its closest neighbours, which incentivises exploration and obtains diverse behaviours. Further, we only use widely applicable general fitness functions based on novelty and solvability to evolve these networks.

We test our methods on a simple *Maze* game as well as *Super Mario Bros*. Our results indicate that our method generates levels significantly faster than both a direct search-based method and an RL-based approach, without the need for game-specific knowledge.

GECCO '22, July 9–13, 2022, Boston, MA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA*, https://doi.org/10.1145/3512290.3528701.

We also show that our method generalises to different sized levels than those trained on, while still mostly generating solvable levels.[1]

2 BACKGROUND

Below we outline two separate approaches that have been previously used to construct PCG systems.

2.1 Evolutionary Search

Genetic algorithms consist of a population of individuals, each possessing a genotype, which can be thought of as the individual's genes. This genotype impacts the phenotype—the manifestation of the genotype in the problem domain. For example, when generating platformer levels, the genotype can be a single integer vector representing the height of platforms across the x-axis [10]. The phenotype, then, is the actual level that has been generated using this specific genotype [11].

High performing individuals are combined using *crossover*, which is the process of combining two parents to form new individuals for the next generation. This new generation is also randomly mutated to facilitate exploration and prevent stagnation. In general, "high performing" is quantified by the *fitness function*. For example, when maximising a function $f(x)$, the fitness could simply be the function value itself.

2.1.1 NeuroEvolution of Augmenting Topologies (NEAT) . NEAT is a method where a genetic algorithm optimises the structure and weights of a neural network [50].

The genetic encoding in NEAT is a linear collection of either node genes, specifying the existence and type (input, hidden or output) of a node, or connection genes, which indicate a connection of a certain weight between two nodes. Mutation can affect both the weights and the structure of the network, either by adding a connection between two nodes, inserting a node between two existing connected nodes, or perturbing a weight's value.

NEAT keeps track of the historic origin of a gene by using the *innovation number*. This enables efficient crossover between different sized parents by first lining up genes with the same history in both parents. Matching genes are then inherited randomly from each parent, whereas genes that occur in only one are inherited from the parent with higher fitness.

NEAT can enable complexity to gradually increase as the search process develops, leading to later generations being more complex than previous ones. Finally, different variations of NEAT exist, most notably HyperNEAT [49], which evolves compositional pattern-producing networks [48]—a type of artificial neural network where each node can use a different activation function—that themselves generate the final network structure. This can lead to larger and more symmetric networks and smaller genomes, although HyperNEAT does not always outperform NEAT [32].

2.1.2 Novelty Search. Novelty search [25] is a different approach to designing the fitness function of a genetic algorithm. Instead of pursuing a higher objective function value, novelty search only judges individuals based on how different or novel they are compared to the current generation and an archive of previously novel individuals. Novelty is defined as the average distance between an individual and its k closest neighbours in behaviour space. Distance can be defined in a domain-agnostic manner, e.g. using a vector norm like absolute difference or Euclidean distance. Domain-specific distance functions can also be used when a more appropriate measure of distance exists. Novelty search encourages agents to pursue novel and diverse behaviours, thereby thoroughly exploring the behaviour space and resulting in diverse individuals [13, 26]. Even though there is no explicit incentive to actually achieve the goal, novelty search can still achieve competitive results, especially in deceptive fitness landscapes when using a traditional objective can result in convergence to local minima [25].

2.2 Reinforcement Learning

An alternate approach to PCG is to use reinforcement learning (RL) to learn a policy that generates new levels [23]. Here the problem is formulated as a Markov decision process $\langle S, A, p, r \rangle$ where S is the set of states, A is the set of available actions, p is the transition dynamics that specify how the environment changes under a given action, and r is the reward function. The aim is to learn a policy $\pi : S \to A$ such that the expected sum of future rewards is maximised [54].

In the context of PCG, Khalifa et al. [23] uses RL to generate levels for 2D tilemap-based games, experimenting with different state and action spaces. For example, in the "turtle" representation, the state is the current level and the coordinate of the current tile under consideration, while A consists of changing the current tile to any other one or moving the agent in one of the four cardinal directions. The "wide" representation, on the other hand, also uses the current level as the state, but actions consist of a coordinate of a tile and a value to change it to. In both cases, the agent is rewarded based on the change that its action causes to the level.

2.3 Related Work

Most procedural level generation approaches can be classified into one of two categories. The first is to directly search in level space. This means that each time the method is run, a new search is carried out for only one level. The genome encoding here is usually direct. Examples of this include the work by Ferreira et al. [10] who search over integer vectors of the same length as the level, Liapis et al. [28, 29] who search over a 2D tilemap representation of the levels, and Cardamone et al. [4] who generate racing game tracks by using a set of control points for Bezier curves as the representation.

The second is to search in generator space. This involves searching for or learning a generator of levels. This generator can be queried to generate a large number of levels by varying its input parameters. The representation used is usually more abstract—for example, the policy of a reinforcement learning agent [23] or the weights of a neural network. Kerssemakers et al. [22] use a genetic algorithm to search for parameter vectors that determine the behaviour of a non-deterministic (and thus reusable) generator.

3 GENERATING DIVERSE LEVELS QUICKLY

Our aim is to develop a method capable of automatic content generation with the following desiderata: the approach must a) be capable of generating levels quickly so that it can be used in real-time games; b) not require any training data, since that would require effort on

[1] Source code is publicly released at https://github.com/Michael-Beukman/PCGNN.

the part of designers to create in the first place; c) use as little game specific information as possible, resulting in a general system; and d) should generate diverse and playable levels.

This section describes our method, Procedural Content Generation using NEAT and novelty search (PCGNN), which satisfies the above requirements, i.e. generating diverse levels in real time without any existing training data or game-specific learning signals.

As previously mentioned, we use NEAT [50] to evolve a neural network that generates levels. This is divided into two parts, training (evolution) and inference (generating levels).

3.1 Generation Process

Given a neural network, we generate a level as follows. We first generate a random 2D array of tiles, and then for each tile (e.g. the question mark in Figure 1) we input the surrounding tiles (those highlighted in Figure 1) into the network using either integer tile values or one-hot encoding. The output is used to predict what tile type should be placed at the original location. To predict boundary tiles, we pad the level with a row or column of −1 on all sides. This method, which is similar to applying a convolution, is used instead of generating the entire level in one step, because the model is then able to generate levels of arbitrary size [41, 45, 61], and make locally consistent choices [18, 53]. We input random noise into the network and perturb all of the inputs to enable the generation of multiple levels, rather than a single deterministic one. This facilitates reuse, allowing the same generator to be used multiple times. This adding of randomness is similar to inputting random noise into a GAN to generate new data [8]. This process is performed sequentially, so the previous predictions become part of the level, and are used by the network as inputs when predicting adjacent tiles.

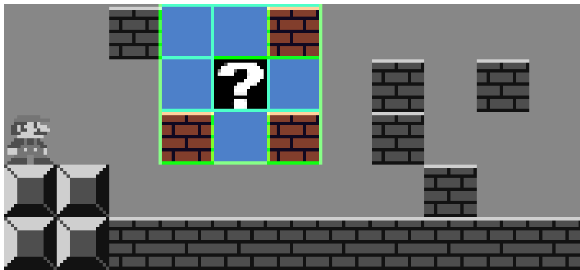


Figure 1: An illustration of the level generation process on *Mario*. The "?" is the current tile to be predicted, whereas the highlighted area is the context that the algorithm uses to predict the center tile.

3.2 Training Process

We use the standard NEAT algorithm [50], and evolve the population using fitness functions that are described in the next section. After evolving the population for a set number of generations, we use the individual with the highest fitness as our final level generator. This can then be used to rapidly produce many new levels, as generation only requires the network to be queried—no learning or searching is performed at inference time.

3.3 Fitness Functions

To evaluate a network, we first generate N levels and use these to compute the fitnesses, which we then average. We use different fitness functions for different purposes, all of which are scaled between 0 and 1 unless otherwise stated. When using multiple fitness functions, we set the final objective as a weighted sum of these fitnesses. This follows work by Gomes et al. [12], who found that this method performs comparably to using multi-objective optimisation techniques, specifically when using the novelty metric.

3.3.1 Solvability. This fitness function determines if the level is solvable, using a search algorithm such as A* [16]. The fitness value is 1 if the level is solvable and 0 otherwise.

3.3.2 Novelty. We also use the novelty metric [25] to evaluate the diversity of level generators. As described above, we do this by generating N levels for each generator. Let G_i denote generator i and L_{in} denote level n from generator i. Then we define the distance function between two generators as

$$D(G_i, G_j) = \frac{1}{N} \sum_{n=1}^{N} d_l(L_{in}, L_{jn}),$$

where $d_l(L_{in}, L_{jn})$ is the distance between two levels. This can be computed using different methods such as Visual Diversity [29], which measures the fraction of non-matching tiles (i.e. the normalised Hamming distance) or perceptual image hashing [15, 36], which gives a low value for images that look similar and a large value for those that look different. The exact function used can have a large effect on the diversity characteristics of the generated levels [40], which we illustrate in the Appendix. Using the above, the novelty score of a generator, with respect to the current population and the novelty archive, is:

$$\text{Novelty}(G_i) = \frac{1}{K} \sum_{k=1}^{K} D(G_i, G_k),$$

where k iterates over the K closest neighbours of G_i, from either the population or the archive. Our archive of previous individuals is formed by randomly adding λ individuals to it at every generation [12] instead of adding only individuals that obtained a high novelty, as in the original work [25]. This is generally preferable, as it leads to more robust parameters and uniform exploration [12].

3.3.3 Intra-Generator Novelty. The above novelty metric rewards novelty between different generators, but the final generator that we use is a single neural network. Thus it is useful to reward generators that can generate multiple diverse levels. To this end, we also use the intra-generator novelty metric, which is similar to the above, but it simply measures the novelty between the N levels generated by one generator, and sets the intra-novelty of this generator to be the average novelty of its N levels. We do not use an archive of previous individuals when computing this value.

3.3.4 Other. Additional fitness functions can be used to generate levels with specific properties (e.g. longer paths or more connected regions [23]), and other feasibility criteria that do not affect solvability, such as not placing enemies in mid-air. These could be used to inject some expert knowledge into the generation or to enforce constraints, but we do not use these in our experiments.

3.4 Summary

Our method uses a learned level generator in the form of a neural network to be able to quickly generate levels after a one-time, offline training period. We evolve this network using NEAT so as to not require any training data, and use general fitness functions, namely solvability and novelty, to provide game-independent learning signals that also incentivise exploration. Algorithm 1 shows pseudocode for the training procedure.

Algorithm 1 PCGNN Training Procedure

```
Require: G ≥ 1 // Number of generations
Require: N ≥ 1 // Number of levels to generate per network
Require: K ≥ 1 // Number of neighbours for novelty
Require: w_f, w_n > 0 // Weighting for standard fitness and novelty
  Pop ← random initial population of networks
  for all generations g ∈ {1, ..., G} do
  -- Generate Levels + Calculate Simple Fitnesses --
    for all net_i ∈ Pop do
      Generate N levels, L_i1, ..., L_iN from network net_i
      // Fitness is e.g. Solvability and/or Intra-Novelty
      Fitness(net_i) ← mean(Fitness(L_ij)), ∀j
    end for
    Set Distance(i, i) = ∞, ∀i
    for all net_i ∈ Pop do
      for all net_j ∈ Pop, j > i do
        // d_l could be e.g. Visual Diversity
        Distance(i, j) ← (1/N) Σ_{n=1}^{N} d_l(L_in, L_jn)
        Distance(j, i) ← Distance(i, j)
      end for
    end for
  -- Calculate Novelty Fitness --
    for all net_i ∈ Pop do
      Sort Distance(i, ·) ascendingly
      Novelty ← (1/K) Σ_{k=1}^{K} Distance(i, k)
      // Linear combination of different fitnesses
      Fitness(net_i) ← w_f · Fitness(net_i) + w_n · Novelty
    end for
    Update population using calculated fitnesses.
  end for
  Return best individual network.
```

4 EXPERIMENTS

Here we detail our experimental setup, the baselines we compare against, and the metrics we use to evaluate the levels.

We consider two 2D tilemap games: firstly, a *Maze* game consisting of "wall" and "empty" tiles, where the objective is to find a path between the top left and bottom right and, secondly, a simplified *Super Mario Bros.*, without powerups and only Goombas as enemies.

4.1 Baselines

We compare PCGNN to several baselines. The first is a genetic algorithm-based approach due to Ferreira et al. [10], which was originally developed for *Mario*. In this method, there is a population for each tile type in the level (e.g. ground, enemies, etc.), and each of these is evolved independently using a genetic algorithm with an entropy [43] or sparseness-based fitness function [7]. For the *Maze*, we simply use the 2D grid directly as the genotype, using partial solvability and entropy as the fitness functions.

Partial solvability is a less sparse version of the solvability fitness function described previously, directly applicable for the *Maze*, and it returns $\frac{1}{3}(\mathbb{1}_{start} + \mathbb{1}_{end} + \mathbb{1}_{connected})$, where $\mathbb{1}_x = 1$ if x is true; 0 otherwise, *start*, *end* mean that the starting and ending tiles are empty, and *connected* is true when the starting and ending tiles are connected by a path of empty tiles. This is to give potentially more guidance to the algorithm than the sparse solvability described above, and was found to perform better for the direct genetic algorithm specifically. Entropy is a fitness function where we split the levels into non-overlapping chunks, calculate the entropy [43] of each chunk, and return the average. The fitness of an individual is then calculated as the distance to the "desired entropy", which is specified by the user [10]. Calculating the entropy of a chunk works as follows: for each tile type t (e.g. wall and empty for the *Maze* game), we count the number of tiles that have that value. From here we can construct a probability distribution and calculate the entropy. For example, let m be the number of tiles in each chunk, then $P_t = \frac{\text{Count}(t)}{m}$ is the probability of tile t, and the entropy for that chunk is defined as

$$H(P) = -\sum_{t=1}^{n} \log_2(P_t) P_t,$$

where n is the number of unique tile types. We normalise the above by dividing by $\log_2(n)$ when $n > 2$.

We also compare how this method performs with the novelty metric as a fitness function, similar to Liapis et al. [29], but without using the two population approach. This method is henceforth referred to as DirectGA.

Our second baseline is Procedural Content Generation via Reinforcement Learning (PCGRL) [23], where the level generation process is modelled as a reinforcement learning problem. For the *Maze* we use reward functions that incentivise solvable levels with path lengths in a certain range, and for *Mario* we reward solvability of the level, feasibly placed enemies as well as having the number of enemies in a certain range. We use the original implementation,[2] and only consider the "turtle" and "wide" representations, as these performed the best in the original work.

The first baseline was chosen to compare against an evolutionary approach that directly searches for levels, as opposed to our method searching in generator space. PCGRL was chosen to specifically compare against another method that learns a level generator and should have a fast generation time.

We do not consider CPPN2GAN [41] or the approach by Volz et al. [61] as baselines, since these methods require existing levels as training data. Similarly, EDRL [45] uses a GAN-based chunk generator trained on existing *Mario* levels, though in principle any parameterised generator could work.

[2] https://github.com/amidos2006/gym-pcgrl

4.2 Metrics

To determine the characteristics of the levels that we generate, we use different metrics described below.

4.2.1 Solvability. Solvability is a relatively simple metric: we determine if the level is solvable using a breadth-first search or A* agent to traverse the level from the starting state to the goal state.

4.2.2 Generation Time. Since one of our goals is to generate levels in real time, we also measure the time it takes each method to generate one level, after (possibly) performing offline training.

4.2.3 Difficulty. We use the leniency [42, 46], and A* difficulty [2] metrics to evaluate the levels based on difficulty. The A* metric measures the number of nodes unnecessarily expanded by the A* algorithm, while leniency calculates how forgiving each obstacle is to a player's mistakes, and averages this across the entire level. For the *Maze*, this measures the fraction of dead ends in the level [2].

4.2.4 Diversity. We use the compression distance [27, 42] metric, particularly the Normal variant as described by Beukman et al. [2], which measures diversity by how much space is saved when compressing two strings (levels) together versus separately. We also use the A* diversity metric [2], which compares the trajectories of an A* agent on pairs of levels—levels that are solved in different ways are marked as diverse

4.3 Implementation details

We use a standard implementation [35] of NEAT to perform the evolutionary process. For a game with n tiles, the network outputs a single tile type at each step. This is effectively an n-class classification problem. For the *Maze*, since there are only two tiles, we use one neuron and a threshold: if the neuron's activation is larger than this threshold, it is a wall, otherwise an empty space. For *Mario*, the network has n output neurons, and the one with the largest activation is chosen as the tile. Each tile is represented as either a binary number (for the *Maze*) or a one-hot encoded vector for *Mario*.

4.3.1 Agents. For *Mario*, we use the updated Mario-AI framework[3] to import our generated levels and use the A* agent by Robin Baumgarten[4] [58, 59] to evaluate them.

Since a level might be solvable in multiple different ways, and the A* agent sometimes performs differently on the same level of *Mario* (due to the relative complexity of the simulation), we run the diversity and difficulty metrics 5 times on each level (for both games) and average the results. For the solvability metric, we also run the agent five times, and if it solved the level in any of these five runs, we label it as solvable. The solvability fitness function uses a less complex but much faster simulation,[5] and we only use the full fidelity simulation when evaluating the levels after training and generation. Conversely, the *Maze* game is simpler, and so we use a standard implementation of A*.

[3]https://github.com/amidos2006/Mario-AI-Framework
[4]https://github.com/amidos2006/Mario-AI-Framework/tree/master/src/agents/robinBaumgarten
[5]https://github.com/amidos2006/gym-pcgrl/blob/master/gym_pcgrl/envs/probs/smb/engine.py

4.4 Experimental Setup

We compare our method of level generation against the aforementioned baselines using the metrics discussed above. For all generation methods, we perform a hyperparameter search and in all cases report the best result obtained.

All level generation experiments are run over five different random seeds with the average and standard deviation reported. Each of these five runs consist of generating 100 different levels, and the metrics of the 100 levels are averaged to obtain a single value for each seed. Since the diversity metrics calculate scores for each pair of levels, we follow Horn et al. [19] and measure the diversity between a group of N levels by calculating the average of the $\frac{N(N-1)}{2}$ pairwise diversity scores.

When reporting metrics, we only use the solvable levels and average over their values to minimise the effect that unsolvable levels have on the metrics. Since the fraction of solvable levels is mostly high, we still average over many levels. We also use a random baseline but this rarely generates solvable levels. Thus, we consider all levels *only* for this baseline.

In the next sections, we use the following names to refer to the specific baselines. See the Appendix for detailed hyperparameters.

PCGNN This is our NEAT and novelty search method.
DirectGA Only applicable to *Mario*, this is the genetic algorithm with the same parameters as in the original work [10].
DirectGA+ The DirectGA method (for both *Maze* and *Mario*), where we perform a hyperparameter search to obtain levels with high solvability, breaking ties based on other metrics, like compression distance.
DirectGA (Novelty) This uses the DirectGA method, but adds in novelty as a fitness function, with uniform weights to all fitness components. For *Mario*, we only consider population sizes and numbers of generations larger than 50, to ensure that the novelty search actually takes effect.
PCGRL (Wide/Turtle) Using PCGRL with the "wide" or "turtle" representations respectively [23].
Random Tiles are selected uniformly at random.

For PCGRL, we attempt to train the method for the same 100 million timesteps as stated by Khalifa et al. [23], but the *Mario* training process was slower than the *Maze* (possibly due to the much larger level sizes), and we only managed to perform about 12 million steps for "wide" and 8 million for "turtle" in 3 days, which are the models we use to report results here. When performing inference for *Mario*, we limit the maximum number of steps per level to 10 000, as without this the PCGRL model sometimes becomes stuck in a loop.

Finally, for both DirectGA approaches, each level is generated from a separate evolution process, starting from a unique initial population. The individual with the highest fitness after G generations is selected as the level. This is repeated 100 times per seed.

4.4.1 Statistical Approach. For the following sections, we use the notation $\mu_{\text{b}}(M)$ to refer to the mean of metric M when using method b, and $\sigma_{\text{b}}(M)$ denotes the standard deviation. All results in the following tables are of the form μ (σ).

Table 1: Train and generation times. Lower is better.

	Maze Time (s)		Mario Time (s)	
	Generation	Train	Generation	Train
PCGNN (Ours)	2.4×10^{-3} (0)	1100.46 (46.41)	0.08 (0.01)	11593 (347)
DirectGA	-	-	**1.78 (0.02)**†	**0.0 (0)**†
DirectGA+	**7.45 (0.18)**†	**0.0 (0)**†	**0.56 (0.01)**†	**0.0 (0)**†
DirectGA (Novelty)	**4.85 (0.05)**†	**0.0 (0)**†	**27.04 (0.49)**†	**0.0 (0)**†
PCGRL (Wide)	**1.62 (2.10)**†	**245775 (6246)**†	**0.98 (1.10)**†	**259200 (0)**†
PCGRL (Turtle)	**6.00 (5.49)**†	**40838 (496)**†	**7.35 (4.11)**†	**259200 (0)**†
Random	0.0 (0)	**0.0 (0)**†	0.0 (0)	**0.0 (0)**†

Our statistical analysis procedure is as follows. We first performed the Kruskal–Wallis test [24] to determine whether a statistical significant difference exists at all. Then, we performed pairwise Mann-Whitney U tests [33] between PCGNN and each baseline's result. This was done because for most of the metrics tested, at least one method failed a Shapiro-Wilk normality test [44] with $p < 0.05$. Finally, since using multiple pairwise tests increases the risk of making a type I error [20, 39], we subsequently perform Bonferroni Error correction [3]. If we find a statistically significant result, we also calculate the Cohen's d value [6], $d = \frac{\mu_{\text{pcgnn}}(M) - \mu_b(M)}{\sigma_{\text{pcgnn}(M)}}$ to measure how large this difference is. In the tables shown in the next section, we use **bold** to denote a statistically significant result ($p < 0.05$) and † to denote a large effect size ($|d| > 0.8$).

All results where we report time were run on similar hardware (details in Appendix), with minimal other processes running to enable a fair comparison.

5 RESULTS

We compare PCGNN against the baselines on generation time in Section 5.1 and on solvability, diversity and difficulty in Sections 5.2, 5.3 and 5.4, respectively. We finally consider generalisability in Section 5.5. Example levels and PCGNN's fitness curves are shown in the Appendix.

5.1 Generation Time

Since one of our goals is to generate levels in a fast, real-time fashion, we compare the generation time of our method to our baselines under the null hypothesis

$$H_0 : \mu_{\text{pcgnn}}(t) \geq \mu_b(t)$$

i.e. that our method is comparable or slower at generating levels than the baselines.

Since PCGNN has a two stage process of evolving the generator and then using the generator to generate levels, we split the total time taken for this into *training* and *testing*. Training is the time required to evolve the generator, which happens once. Generation then refers to querying this generator. The DirectGA approaches generate levels as needed, and so have 0 training time.

We use a one-sided Mann-Whitney U test for generation time and a two-sided test for training time. Table 1 presents the results.

We thus reject the null hypothesis that our method's generation time is comparable or slower than our baselines (except for the Random method, which does not perform any computation). We see a large effect size, and at least an order of magnitude improvement in generation speed. For training time, we find a statistically significant difference compared to all other methods (using a two-sided test). This is to be expected, since the DirectGA method requires no training time and PCGRL requires substantially more.

PCGRL has quite a large variance in generation times, and we hypothesise that this is because it generates a level until a specified reward threshold is met, instead of iterating for a fixed number of iterations like PCGNN and DirectGA.

We do not parallelise the fitness function calculations for either DirectGA or PCGNN, although doing so could improve performance, notably generation time for DirectGA. This would largely depend on the number of cores the generating machine has, and many cores are often not a given on a user's machine [51]. Similarly, we only take the top individual after performing the evolution—it would be more efficient to take more individuals from the final generation, but they might lack in diversity or feasibility [14].

5.2 Solvability

Since we can generate levels quickly, we next investigate the quality of these levels, starting with solvability. Here we compare solvability scores between our method and the baselines, with the null hypothesis that our method has the same mean solvability as the other methods. Table 2 shows these results.

Table 2: Solvability. Higher is better.

	Maze Solvability	Mario Solvability
PCGNN (Ours)	1.0 (0)	0.98 (0.02)
DirectGA	-	1.0 (0)
DirectGA+	1.0 (0)	0.99 (0.01)
DirectGA (Novelty)	**0.92 (0.04)**†	1.0 (0)
PCGRL (Wide)	1.0 (0)	0.74 (0.03)
PCGRL (Turtle)	0.95 (0.08)	0.70 (0.02)
Random	$\mathbf{2.0 \times 10^{-3}}$ **(0)**†	0.08 (0.03)

We find no statistically significant difference between our solvability, the solvability of PCGRL and that of DirectGA+ for the *Maze*. For *Mario*, we find no statistically significant difference between PCGNN's solvability and that of the baselines.

PCGRL's solvability on *Mario* is much lower than on *Maze*, possibly due to training for fewer timesteps, and increasing the training budget could improve this. Our solvability is thus perfect for *Maze*, and still very high on *Mario*. The above result, coupled with our fast generation times, indicates that we can generate a larger number of solvable levels than the other methods in the same amount of time.

5.3 Diversity

We now compare the diversity of the generated levels using the compression distance and A* diversity metrics. The null hypothesis is that the diversity of PCGNN is comparable to the baselines. Results are shown in Table 3, where we see that PCGNN's A* diversity is comparable to DirectGA+ for *Mario*, but rather different from both PCGRL representations and quite different to all methods for *Maze*. Because of the high variance in PCGNN's compression

Table 3: Diversity metrics. Higher is better.

	Maze		Mario	
	Compression Distance	A* Diversity	Compression Distance	A* Diversity
PCGNN (Ours)	0.488 (0.002)	0.13 (0.17)	0.45 (0.10)	0.38 (0.11)
DirectGA	-	-	0.46 (0)	**0.10 (0.01)**†
DirectGA+	0.493 (0.002)	0.41 (0.01)	0.55 (0)	0.33 (0)
DirectGA (Novelty)	0.494 (0.002)	0.40 (0.01)	0.44 (0.01)	0.20 (0.01)
PCGRL (Wide)	**0.525 (0.021)**†	0.43 (0.01)	0.56 (0)	**0.55 (0)**†
PCGRL (Turtle)	**0.513 (0.006)**†	0.43 (0.01)	0.56 (0)	**0.55 (0)**†
Random	**0.494 (0.001)**†	0.00 (0.01)	0.47 (0)	**0.72 (0.02)**†

Table 4: Difficulty. The optimal value usually depends on the player's skill and designer's intentions.[6]

	Maze		Mario	
	Leniency	A* Difficulty	Leniency	A* Difficulty
PCGNN (Ours)	0.70 (0.08)	0.06 (0.08)	0.17 (0.23)	0.24 (0.06)
DirectGA	-	-	**1.0 (0)**†	0.27 (0)
DirectGA+	**0.60 (0.01)**†	0.16 (0.02)	0.31 (0.01)	**0.14 (0.01)**†
DirectGA (Novelty)	**0.59 (0.01)**†	0.16 (0.01)	0.78 (0.04)	0.24 (0.01)
PCGRL (Wide)	0.64 (0.04)	0.21 (0.01)	0.47 (0.01)	0.29 (0.01)
PCGRL (Turtle)	0.71 (0.02)	0.18 (0.03)	0.47 (0.01)	0.28 (0.01)
Random	2.3×10^{-3} (0)	0.52 (0.02)	0.14 (0)	**0.88 (0.02)**†

distance for *Mario*, we find no statistically significant difference between our method and the baselines.

The A* diversity metric evaluates DirectGA's *Mario* levels (which are relatively flat without jumps or platforms) as quite similar, since the same rough trajectory solves most levels. PCGRL's levels, on the other hand, require substantially different trajectories in general, indicating that the levels cannot all be solved using the same path. PCGNN is somewhere in between, indicating that different trajectories and strategies are required, but the difference, on average, is not as large as PCGRL.

We also note that using novelty for the DirectGA does not improve the diversity metrics (compared to DirectGA+), potentially indicating a mismatch between what novelty rewards and what the metrics measure.

5.4 Difficulty

Next we investigate the difficulty of our levels, as measured by the leniency [42, 46] and A* difficulty [2] metrics. These metrics both attempt to measure the abstract notion of "difficulty", but they do so in different ways. Further, levels with higher leniency correspond to levels with lower A* difficulty and vice versa. We again use the null hypothesis that our method has the same mean difficulty as our baselines.

Results in Table 4 show that for the *Maze*, our method generates quite lenient levels, and this is confirmed by the lower value of the A* difficulty metric. For *Mario*, our levels have quite low leniency, but this varies drastically. We therefore only find a statistically significant difference between PCGNN and the DirectGA, which generates very flat levels and thus has relatively high leniency.

5.5 Generalisability

This experiment asks the question: "after we have learnt enough to generate one level of size Y, how long does it take to generate a level of size $X \neq Y$?" We hypothesise that PCGNN will be able to generate levels of arbitrary size quickly, without any retraining, as we use a size-agnostic generation method. We test this claim by taking an evolved generator (that was trained only to generate levels of size 14×14), and using it to generate levels of different sizes. We compare this against the DirectGA method in Figure 2.

We find that PCGNN's generation speed is still substantially higher than the DirectGA's (for both *Mario* and *Maze*), even as we increase the level size. PCGNN generates *Maze* levels that have 2000^2 tiles faster than DirectGA generates levels with 100^2 tiles. In the *Maze* domain, our method achieves perfect solvability for all tested sizes, whereas the direct genetic algorithm's solvability decreases drastically as the level size increases. One potential explanation is that we keep the population size and number of generations constant as the level size increases; prior work has shown that standard genetic algorithms do not always perform very well with high dimensional problems [31, 62]. For *Mario*, we see a slight downward trend in solvability as the level size increases, whereas DirectGA remains constant. This DirectGA consistency is because the initial level, instead of being random as in the *Maze* game, is already solvable without any extra input from the genetic algorithm. Again, even with some loss of solvability as the level width increases, our fast generation time can ameliorate this issue.

Since PCGRL relies on a specific input size (as it uses a neural network that can only process a fixed size input), it cannot generalise to different sized levels without an expensive retraining.

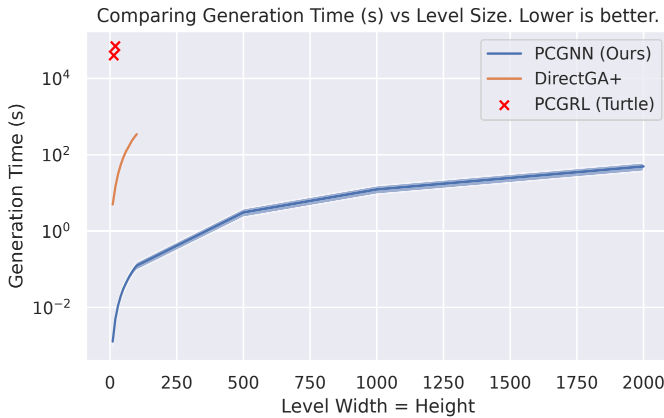
The above results indicate that PCGNN is well-suited to generalising to larger levels without the need for retraining. Thus, even when the fitness function used is expensive, or we require many generations or a large population size, all of this computation can be performed offline without negatively impacting the generation time. On the other hand, DirectGA's generation time is directly affected by these factors, making it increasingly unsuited for real-time generation, as shown in Figure 3.

6 DISCUSSION AND FUTURE WORK

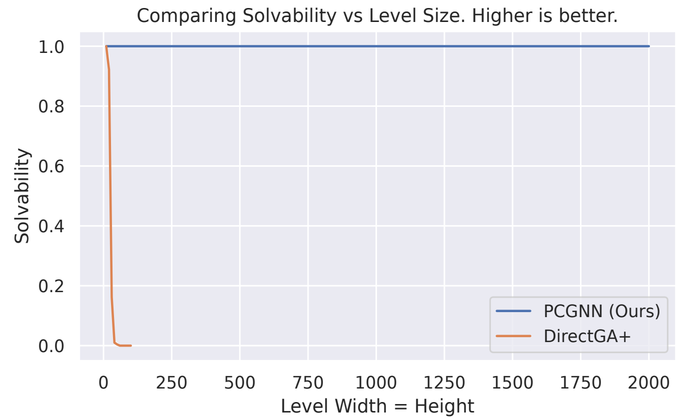
The method introduced here follows a recent trend focusing on level generators, which can be applied in real time after an offline training period [23, 45] instead of searching for the levels directly. We demonstrate several advantages over these works, namely an even faster generation time, no need for training data or game-specific reward functions, and the ability to generalise to different sized levels without retraining.

Limitations of the proposed method include generation time still scaling linearly with the level size, possibly leading to infeasible generation times for massive levels, although this performed well for the tested sizes. Although we demonstrated comparable metric scores to the baselines, the sequential level generation process could still limit the characteristics of the levels by not taking into account global information. Further, while we do not require hand-specified

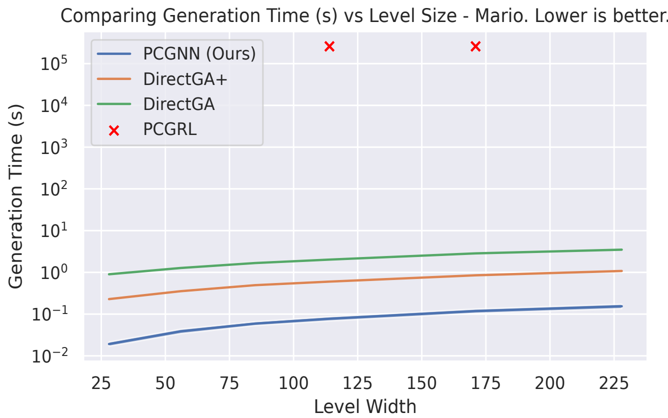
[6]The difference between the leniency of the Random baseline and PCGNN is not statistically significant. The reason for this is that there were ties in the leniency values (multiple seeds had 0 leniency), leading to the asymptotic Mann-Whitney U calculation being used over the exact version [33]. The normal approximation, while decent, is not perfect for a small sample size of 5 [1]. This, combined with the Bonferonni correction, leads to a p value of 0.055, slightly above our threshold of 0.05.



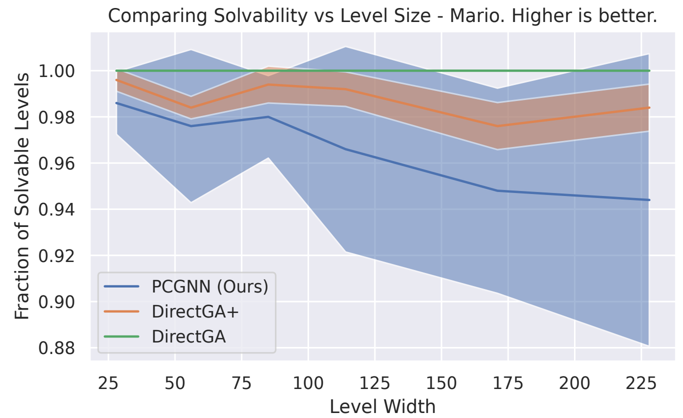
(a) A comparison of generation times on the *Maze* domain (lower is better) on a log scale.



(b) A comparison of solvability with an increase in *Maze* sizes (higher is better).



(c) A comparison of generation times on *Mario* (lower is better) on a log scale.



(d) A comparison of solvability with an increase in *Mario* sizes (higher is better). Note that the y-axis ranges from 0.88 to 1.

Figure 2: Metrics for *Maze* (top row) and *Mario* (bottom row) levels of different sizes. Standard deviation is indicated by the shaded regions. For (a) and (c) we only plot two PCGRL points, since training for larger levels was prohibitively expensive.

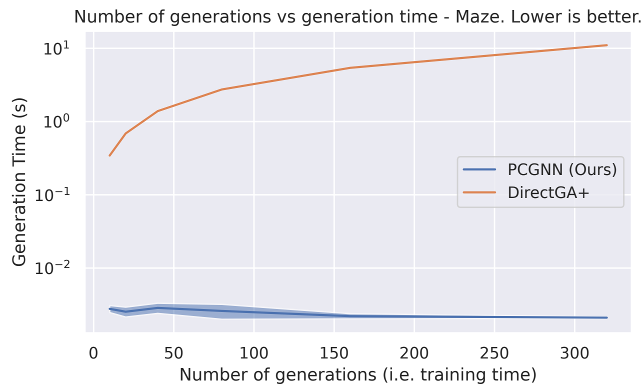


Figure 3: Comparing the effect of increasing the number of generations on generation time in the *Maze* domain with standard deviation shaded.

fitness functions, the generated levels without these elements might not be desired in some cases. The *Mario* levels are also not very similar to the original game, but this was not our goal. More specific domain-knowledge can be incorporated, though, by changing the fitness function.

Future work could include analysing the effects of the different hyperparameters of our method (e.g. context or prediction sizes), or attempting to make the method more configurable during runtime. This would allow users to specify desired characteristics, similar to specifying the level size dynamically as we do here. Additional work could also investigate different novelty search distance functions (with a potential focus on simulation-based ones such as the A* diversity [2]), or attempt to apply multi-objective optimisation [56] or illumination algorithms [37] instead of naïvely optimising the sum of the different fitnesses. Another option is exploring the generation of endless levels, where the next parts of the level are generated as the player proceeds (possibly using the player's behaviour as input to the network to generate adaptive levels) [45].

7 CONCLUSION

We introduced PCGNN, a NEAT and novelty search-based level generation approach that (1) does not require any training data; (2) requires minimal game-specific knowledge; (3) can be generally applied; and (4) generate levels of arbitrary size rapidly after an offline training period. We compare this method against a direct genetic algorithm and a reinforcement learning-based approach that learns a policy instead of directly searching for a level. Our method performs comparably to the baselines in terms of solvability, difficulty and diversity, while generating levels significantly faster. Most importantly, our method requires no hand-engineered rewards or knowledge of game mechanics such as enemies. We believe this generality is an important step towards the widespread adoption of procedural content generation in a variety of contexts.

ACKNOWLEDGMENTS

This work is based on the research supported wholly by the National Research Foundation of South Africa (Grant UID 133358).

Computations were performed using High Performance Computing infrastructure provided by the Mathematical Sciences Support unit at the University of the Witwatersrand.

We thank the reviewers for their helpful and insightful comments, which helped to strengthen the final version of this paper.

REFERENCES

[1] Carine A Bellera, Marilyse Julien, and James A Hanley. 2010. Normal approximations to the distributions of the Wilcoxon statistics: accurate to what N? Graphical insights. *Journal of Statistics Education* 18, 2 (2010).

[2] Michael Beukman, Steven James, and Christopher W. Cleghorn. 2022. Towards Objective Metrics for Procedurally Generated Video Game Levels. *CoRR* abs/2201.10334 (2022). arXiv:2201.10334 https://arxiv.org/abs/2201.10334

[3] Carlo Bonferroni. 1936. Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze* 8 (1936), 3–62.

[4] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. 2011. Interactive Evolution for the Procedural Generation of Tracks in a High-End Racing Game. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (Dublin, Ireland) *(GECCO '11)*. Association for Computing Machinery, New York, NY, USA, 395–402. https://doi.org/10.1145/2001576.2001631

[5] Karl Cobbe, Oleg Klimov, Christopher Hesse, Taehoon Kim, and John Schulman. 2019. Quantifying Generalization in Reinforcement Learning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1282–1289. http://proceedings.mlr.press/v97/cobbe19a.html

[6] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Academic press.

[7] Michael Cook and Simon Colton. 2011. Multi-faceted evolution of simple arcade games. In *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011, Seoul, South Korea, August 31 - September 3, 2011*, Sung-Bae Cho, Simon M. Lucas, and Philip Hingston (Eds.). IEEE, 289–296. https://doi.org/10.1109/CIG.2011.6032019

[8] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. 2018. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine* 35, 1 (2018), 53–65.

[9] ESA. 2021. 2021 Essential Facts about the video game industry. https://www.theesa.com/resource/2021-essential-facts/

[10] Lucas Ferreira, Leonardo Pereira, and Claudio Toledo. 2014. A Multi-Population Genetic Algorithm for Procedural Generation of Levels for Platform Games. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation* (Vancouver, BC, Canada) *(GECCO Comp '14)*. Association for Computing Machinery, New York, NY, USA, 45–46. https://doi.org/10.1145/2598394.2598489

[11] David E Goldberg. 1989. *Genetic algorithms in search*. Addison Wesley Publishing Co. Inc., Chapter 1.

[12] Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. 2015. Devising effective novelty search algorithms: A comprehensive empirical study. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 943–950. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.717.6684&rep=rep1&type=pdf

[13] Jorge Gomes, Paulo Urbano, and Anders Lyhne Christensen. 2013. Evolution of swarm robotics systems with novelty search. *Swarm Intelligence* 7, 2 (2013), 115–144.

[14] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2019. Procedural Content Generation through Quality Diversity. In *IEEE Conference on Games, CoG 2019, London, United Kingdom, August 20-23, 2019*. IEEE, 1–8. https://doi.org/10.1109/CIG.2019.8848053

[15] Azhar Hadmi, William Puech, Brahim Ait Es Said, and Abdellah Ait Ouahman. 2012. Perceptual image hashing. In *Watermarking-Volume 2*. IntechOpen.

[16] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[17] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9, 1 (2013), 1–22.

[18] Amy K Hoover, Julian Togelius, and Georgios N Yannakis. 2015. Composing video game levels with music metaphors through functional scaffolding. In *First computational creativity and games workshop. ACC*.

[19] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. 2014. A comparative evaluation of procedural level generators in the Mario AI framework. In *Foundations of Digital Games 2014, Ft. Lauderdale, Florida, USA (2014)*. Society for the Advancement of the Science of Digital Games, 1–8.

[20] Mohieddin Jafari and Naser Ansari-Pour. 2019. Why, when and how to adjust your P values? *Cell Journal (Yakhteh)* 20, 4 (2019), 604.

[21] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. 2018. Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729* (2018).

[22] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios Yannakakis. 2012. A procedural procedural level generator generator. In *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*. 335–341. https://doi.org/10.1109/CIG.2012.6374174

[23] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. PCGRL: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 16. 95–101.

[24] William H Kruskal and W Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952), 583–621.

[25] Joel Lehman and Kenneth Stanley. 2011. Abandoning Objectives: Evolution Through the Search for Novelty Alone. *Evolutionary computation* 19 (06 2011), 189–223. https://doi.org/10.1162/EVCO_a_00025

[26] Joel Lehman and Kenneth O Stanley. 2011. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 211–218.

[27] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul MB Vitányi. 2004. The similarity metric. *IEEE Transactions on Information Theory* 50, 12 (2004), 3250–3264.

[28] Antonios Liapis, Georgios Yannakakis, and Julian Togelius. 2013. Enhancements to constrained novelty search: two-population novelty search for generating game content. In *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference*. 343–350. https://doi.org/10.1145/2463372.2463416

[29] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2015. Constrained Novelty Search: A Study on Game Content Generation. *Evolutionary computation* 23, 1 (March 2015), 101–129. https://doi.org/10.1162/EVCO_a_00123

[30] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N Yannakakis, and Julian Togelius. 2020. Deep learning for procedural content generation. *Neural Computing and Applications* (2020), 1–19.

[31] Yong Liu, Xin Yao, Qiangfu Zhao, and Tetsuya Higuchi. 2001. Scaling up fast evolutionary programming with cooperative coevolution. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, Vol. 2. Ieee, 1101–1108.

[32] Jessica Lowell, Sergey Grabkovsky, and Kir Birger. 2011. Comparison of NEAT and HyperNEAT Performance on a Strategic Decision-Making Problem. In *Fifth International Conference on Genetic and Evolutionary Computing, ICGEC 2011, Kinmen, Taiwan / Xiamen, China, August 29 - September 1, 2011*, Junzo Watada, Pau-Choo Chung, Jim-Min Lin, Chin-Shiuh Shieh, and Jeng-Shyang Pan (Eds.). IEEE Computer Society, 102–105. https://doi.org/10.1109/ICGEC.2011.33

[33] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

[34] David Maung and Roger Crawfis. 2015. Applying formal picture languages to procedural content generation. In *2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES)*. IEEE, 58–64.

[35] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher de Silva. [n. d.]. *neat-python*. https://github.com/CodeReclaimers/neat-python

[36] Vishal Monga and Brian L Evans. 2006. Perceptual image hashing via feature points: performance evaluation and tradeoffs. *IEEE transactions on Image Processing* 15, 11 (2006), 3452–3465.

[37] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *CoRR* abs/1504.04909 (2015). arXiv:1504.04909 http://arxiv.org/abs/1504.04909

[38] David Pinelle, Nelson Wong, and Tadeusz Stach. 2008. Heuristic evaluation for games: usability principles for video game design. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1453–1462.

[39] Arnu Pretorius, Elan Van Biljon, Benjamin van Niekerk, Ryan Eloff, Matthew Reynard, Steven D. James, Benjamin Rosman, Herman Kamper, and Steve Kroon. 2019. If dropout limits trainable depth, does critical initialisation still matter? A large-scale statistical analysis on ReLU networks. *CoRR* abs/1910.05725 (2019). arXiv:1910.05725 http://arxiv.org/abs/1910.05725

[40] Mike Preuss, Antonios Liapis, and Julian Togelius. 2014. Searching for good and diverse game levels. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26-29, 2014*. IEEE, 1–8. https://doi.org/10.1109/CIG.2014.6932908

[41] Jacob Schrum, Vanessa Volz, and Sebastian Risi. 2020. CPPN2GAN: Combining Compositional Pattern Producing Networks and GANs for Large-Scale Pattern Generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 139–147. https://doi.org/10.1145/3377930.3389822

[42] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'neill. 2012. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 304–311.

[43] Claude E Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[44] S Shaphiro and M Wilk. 1965. An analysis of variance test for normality. *Biometrika* 52, 3 (1965), 591–611.

[45] Tianye Shu, Jialin Liu, and Georgios N. Yannakakis. 2021. Experience-Driven PCG via Reinforcement Learning: A Super Mario Bros Study. In *2021 IEEE Conference on Games (CoG)*. IEEE.

[46] Gillian Smith, Jim Whitehead, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha. 2010. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on computational intelligence and AI in games* 3, 1 (2010), 1–16.

[47] William M Spears et al. 1995. Adapting crossover in evolutionary algorithms.. In *Evolutionary programming*. 367–384.

[48] Kenneth O. Stanley. 2007. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines* 8, 2 (June 2007), 131–162. https://doi.org/10.1007/s10710-007-9028-8

[49] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artif. Life* 15, 2 (2009), 185–212. https://doi.org/10.1162/artl.2009.15.2.15202

[50] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* 10, 2 (June 2002), 99–127. https://doi.org/10.1162/106365602320169811

[51] Steam. 2021. Steam Hardware & Software Survey: September 2021. https://store.steampowered.com/hwsurvey/. [Online; accessed 25-October-2021].

[52] Nathan Sturtevant and Matheus Ota. 2018. Exhaustive and semi-exhaustive procedural content generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 14.

[53] 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). *IEEE Transactions on Games* 10, 3 (2018), 257–270. https://doi.org/10.1109/TG.2018.2846639

[54] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[55] Gilbert Syswerda. 1993. Simulated crossover in genetic algorithms. In *Foundations of genetic algorithms*. Vol. 2. Elsevier, 239–255.

[56] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. 1996. Multi-Objective Optimization by Genetic Algorithms: A Review. In *Proceedings of 1996 IEEE International Conference on Evolutionary Computation, Nayoya University, Japan, May 20-22, 1996*, Toshio Fukuda and Takeshi Furuhashi (Eds.). IEEE, Japan, 517–522. https://doi.org/10.1109/ICEC.1996.542653

[57] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. 2013. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Artificial and Computational Intelligence in Games*, Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius (Eds.). Dagstuhl Follow-Ups, Vol. 6. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 61–75. https://doi.org/10.4230/DFU.Vol6.12191.61

[58] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. 2010. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*. IEEE, 1–8.

[59] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. 2013. The mario ai championship 2009-2012. *AI Magazine* 34, 3 (2013), 89–92.

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

[61] 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 *Proceedings of the Genetic and Evolutionary Computation Conference*. 221–228.

[62] Xin Yao and Yong Liu. 1998. Scaling up evolutionary programming algorithms. In *International Conference on Evolutionary Programming*. Springer, 103–112.

APPENDIX

Here we list the detailed hyperparameters that were used in the above experiments. Implementation details can be found in our source code located at https://github.com/Michael-Beukman/PCGNN.

Hardware Details

We ran the experiments on Intel i9-10940X CPUs, and PCGRL's training and inference was performed on NVIDIA RTX3090 GPUs.

Hyperparameter Search

We performed a small grid search over all variables for each method with between 2 and 4 values per variable. The hyperparameter set was chosen to maximise solvability, with compression distance used as a tie-breaker. Tables 5 and 6 contain the values we tested.

For PCGRL, we only searched over one variable for *Mario*; namely, how much to weigh how far the agent progressed in the level. We specifically searched over the values $[0, 2, 5, 8, 10]$ and found that the default value of 5 performed the best.

The following sections contain some more descriptions of the baselines, as well as the actual hyperparameters.

DirectGA

For the DirectGA, we used 2 point crossover, flattening the 2D array in the *Maze* case, and 1 point crossover for *Mario* with roulette selection and a mutation probability of 20%. We used these, as they were unspecified by Ferreira et al. [10], simple to implement, relatively standard [11] and the crossover operations were shown to perform relatively well [47, 55].

We also make the fitness function simply the inverse of the absolute difference between the actual and desired levels of entropy, ensuring that values do not become larger than 10. This is the fitness we then maximise.

Table 5: The hyperparameters we searched over for PCGNN. $[0, 1]$ **means the search was performed over the interval.**

	Maze	Mario
Context Size	1	1
Predict Size	1	1, 2, 3
Number of Random Variables	4	4
Random Perturb Size	[0, 1]	0
Number of Generations	10, 50, 100, 200	10, 20, 50, 150, 300
Population Size	20, 50, 100	20, 50, 100
Number of Levels	15, 24	3, 5, 6, 15
λ	0, 1, 2, 3, 4, 5, 6	0, 1
One Hot Inputs	False	False, True
Novelty Distance Function	Hashing (Average), Hashing (Perceptual Simple), Hashing (Perceptual), Hashing (Wavelet), Visual Diversity, Visual Diversity Reachable	Visual Diversity
Intra-Novelty Weight	[0, 1]	1
Novelty Weight	[0, 1]	1
Solvability Weight	[0, 1]	1, 2, 3, 4, 6, 10
Use Entropy Fitness	False, True	False
Use Intra-Novelty Fitness	False, True	True
Use Solvability Fitness	False, True	True

Table 6: The hyperparameters we searched over for DirectGA

	Mario	Maze
Population Size	10, 50, 100	10, 50, 100
Number of Generations	10, 50, 100	10, 50, 100
Desired Entropy	0, 0.5, 1.0	0, 0.5, 1
Desired Sparseness Enemies	⟨Desired Entropy⟩	-
Desired Sparseness Coins	⟨Desired Entropy⟩	-
Desired Sparseness Blocks	⟨Desired Entropy⟩	-
Entropy Block Size	20, 114	-
Enemies Block Size	20	-
Coin Block Size	10	-
Blocks Block Size	10, 40	-
Ground Maximum Height	2, 5	-
Coin Maximum Height	⟨Ground Maximum Height⟩	-
Use Novelty Fitness	True	False, True
Use Solvability Fitness	False	False, True

The specific hyperparameters used are shown in Tables 7 and 8. For these tables, *DE* is Desired Entropy and *PSolvability* is Partial Solvability.

Table 7: *Maze* hyperparameters for DirectGA

	DirectGA+	DirectGA Novelty
Population Size	100	50
Number of Generations	100	100
Fitness	Entropy(DE=1) × 0.5 PSolvability() × 0.5	Entropy(DE=0) × 0.33 PSolvability() × 0.33

For the DirectGA with novelty on *Maze*, we additionally use Visual Diversity, $\lambda = 1$ and 15 neighbours, with a weight of 0.33.

Table 8: *Mario* hyperparameters for DirectGA. Explanations of these parameters are provided by Ferreira et al. [10].

	DirectGA+	DirectGA Novelty	DirectGA
Population Size	10	100	20
Number of Generations	50	100	100
Desired Entropy	0.5	0.0	0.0
Desired Sparseness Enemies	0.5	0.0	0.0
Desired Sparseness Coins	0.5	1.0	1.0
Desired Sparseness Blocks	0.5	0.5	0.5
Entropy Block Size	20	114	114
Enemies Block Size	20	20	20
Coin Block Size	10	10	10
Blocks Block Size	40	10	10
Ground Maximum Height	2	2	2
Coin Maximum Height	2	2	2

For the DirectGA with novelty on *Mario*, we additionally use Visual Diversity, 6 neighbours in the novelty calculation and $\lambda = 1$. This was done for each individual element (ground, enemies, coins, etc.) and novelty had the same weight as the normal fitness function.

PCGRL

For PCGRL, as previously mentioned, we trained for 100 million timesteps for *Maze*, and 8 million and 12 million for *Mario* Turtle and Wide respectively.

The reward function used for the *Maze* was similar to the one used by Khalifa et al. [23] in that we reward having only one region, but we differ in that we reward paths only between the start and goal tiles, with lengths between 20 and 80. We weight these different effects with a ratio of 5:2.

For *Mario*, the exact reward function we use can be found in the PCGRL Github repository,[7] and this rewards levels that have the following characteristics:

- Enemies are directly above solid tiles and are not floating in the air.
- Tubes are not disjoint.
- There are between 10 and 30 enemies per level.
- At least 60% of the level is empty.
- The level has minimal noise, i.e. tiles of the same type are usually grouped together.
- The level requires at least 20 jumps to solve.
- The distance required to jump is low.
- The level is solvable.

[7] https://github.com/amidos2006/gym-pcgrl/blob/master/gym_pcgrl/envs/probs/smb_prob.py

PCGNN

The hyperparameters for PCGNN are shown in Table 9.

Examples

Example levels from the tested methods are shown in Figures 4 and 5.

Additional Experiments

Here we show some additional results.

Novelty Distance Functions. Firstly, there are many different distance functions we can use for the novelty fitness calculation. Some of the ones we tried are listed below. All image hashing methods used the ImageHash library.[8]

Euclidean The Euclidean distance between levels with n unique tiles, where each tile is an integer from 0 to $n - 1$.

Hashing (Perceptual Simple) Simple Perceptual Image Hashing.

Hashing (Average) Average Image Hashing.

Visual Diversity Hamming distance, i.e. fraction of tiles that are different.

Hashing (Perceptual) Perceptual Image Hashing.

Hashing (Wavelet) Wavelet Image Hashing.

Some distance functions are only applicable for the *Maze*, as some aspects (like pathfinding) were much faster on this domain, making these functions feasible.

Visual Diversity Reachable Visual Diversity, but all tiles that are not reachable from the starting tile are set to a wall.

JS This takes the coordinates of each reachable tile, and creates a probability distribution for each level. The final distance is then the Jensen-Shannon divergence between these two distributions.

Path Comparing the shortest paths from start to end in each level. The paths are compared using the average Manhattan distance between corresponding steps in the trajectories.

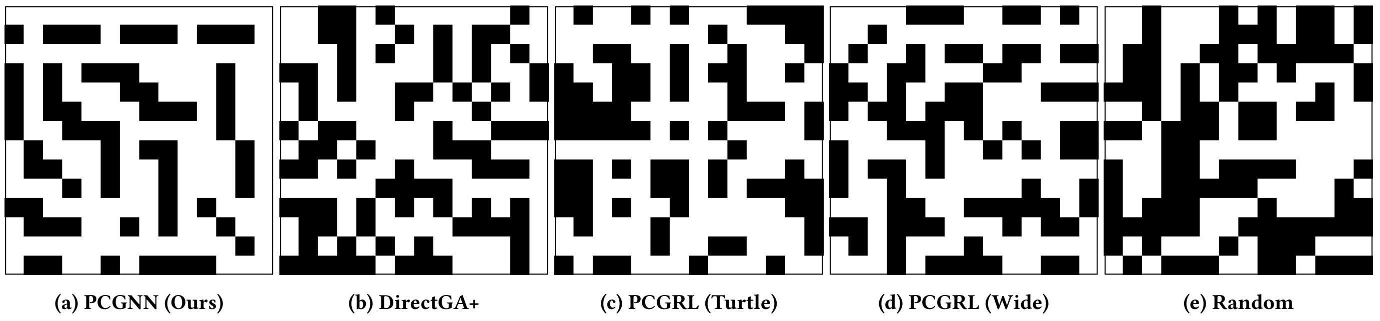
Window This considers all reachable tiles in the level, and sets the distance to the average visual diversity of 3×3 blocks centred at each location.

Window (V2) This is similar to the above, but instead of taking the reachable tiles, uses only the shortest path as a trajectory.

Results when using different distance functions for *Maze* and *Mario* are shown in Tables 10 and 11 respectively. Example levels for each of these functions are also shown in Figures 6 and 7.

PCGNN Fitness Plots. Figure 8 shows the different fitness functions and their value over the evolution process for PCGNN.

[8] https://github.com/JohannesBuchner/imagehash



(a) PCGNN (Ours) (b) DirectGA+ (c) PCGRL (Turtle) (d) PCGRL (Wide) (e) Random

Figure 4: *Maze*: Example levels.

(a) PCGNN (Ours)

(b) DirectGA+

(c) DirectGA

(d) PCGRL (Turtle)

(e) PCGRL (Wide)

(f) Random

Figure 5: *Mario*: Example levels.

Table 9: Hyperparameters for PCGNN

	Maze	Mario
Context Size	1	1
Predict Size	1	1
Number of Random Variables	4	4
Padding	-1	-1
Random Perturb Size	0.1565	0.0
Number of Generations	200	150
Population Size	50	100
Number of Levels	24	6
Number of Neighbours for Novelty Calculation	15	15
λ	0	0
Novelty Distance Function	Visual Diversity, only on reachable tiles	Visual Diversity
Fitness	Novelty() × 0.399 Solvability() × 0.202 *Intra-Novelty*(neighbours=10) × 0.399	Novelty() × 0.25 Solvability()× 0.50 *Intra-Novelty*(neighbours=2) × 0.25

Table 10: *Maze*: Comparing a collection of different distance functions for the novelty and intra-novelty fitness functions. Blue denotes the maximum and green denotes the minimum per column. This table shows $\mu(\sigma)$. The generation time is also not dependent on the distance function – so even if the fitness takes a long time to compute, generation is still fast.

Distance Function	Generation Time (s)	Train Time (s)	Solvability	Compression Distance	A* Diversity	Leniency	A* Difficulty
Path	0.002 (0.0)	14528 (384)	1.0 (0)	0.460 (0.004)	0.001 (0.0)	0.86 (0.03)	0.0 (0.0)
Window (V2)	0.002 (0.0)	14331 (357)	0.95 (0.09)	0.466 (0.014)	0.10 (0.20)	0.79 (0.11)	0.04 (0.08)
JS	0.003 (0.0)	3441 (50)	1.0 (0)	0.490 (0.007)	0.0 (0)	0.62 (0.04)	0.0 (0)
Visual Diversity Reachable	0.003 (0.0)	1107 (46)	1.0 (0)	0.488 (0.002)	0.13 (0.17)	0.70 (0.08)	0.06 (0.08)
Hashing (Wavelet)	0.002 (0.0)	8156 (54)	1.0 (0)	0.498 (0.017)	0.24 (0.19)	0.75 (0.11)	0.06 (0.05)
Euclidean	0.002 (0.0)	794 (40)	1.0 (0)	0.495 (0.001)	0.0 (0)	0.62 (0.01)	0.0 (0)
Hashing (Perceptual)	0.003 (0.0)	3373 (62)	0.99 (0.02)	0.390 (0.087)	0.06 (0.08)	0.98 (0.02)	0.02 (0.02)
Window	0.002 (0.0)	11546 (172)	0.99 (0.01)	0.481 (0.008)	0.17 (0.18)	0.71 (0.03)	0.07 (0.07)
Visual Diversity	0.002 (0.0)	739 (24)	1.0 (0)	0.493 (0.002)	0.07 (0.14)	0.66 (0.07)	0.02 (0.04)
Hashing (Average)	0.002 (0.0)	1860 (24)	1.0 (0)	0.495 (0.001)	0.0 (0)	0.62 (0.01)	0.0 (0)
Hashing (Perceptual Simple)	0.003 (0.0)	2392 (55)	1.0 (0)	0.489 (0.014)	0.0 (0)	0.68 (0.10)	0.0 (0)

Table 11: *Mario*: Comparing a collection of different distance functions for the novelty and intra-novelty fitness functions. Blue denotes the maximum and green denotes the minimum per column. This table shows $\mu(\sigma)$. The generation time is also not dependent on the distance function – so even if the fitness takes a long time to compute, generation is still fast. The A* diversity metric value for Visual Diversity is slightly different from that shown in the main text due to some stochasticity in the behaviour of the A* agent.

Distance Function	Generation Time (s)	Train Time (s)	Solvability	Compression Distance	A* Diversity	Leniency	A* Difficulty
Hashing (Perceptual)	0.07 (0.01)	12929 (80)	0.92 (0.07)	0.35 (0.23)	0.46 (0.07)	0.14 (0.22)	0.20 (0.02)
Visual Diversity	0.08 (0.01)	11508 (279)	0.98 (0.02)	0.45 (0.10)	0.39 (0.11)	0.17 (0.23)	0.24 (0.06)
Hashing (Average)	0.07 (0.01)	12273 (325)	0.75 (0.18)	0.50 (0.06)	0.58 (0.10)	0.23 (0.22)	0.33 (0.08)
Hashing (Wavelet)	0.07 (0.01)	15898 (305)	0.74 (0.21)	0.44 (0.14)	0.56 (0.11)	0.41 (0.37)	0.39 (0.24)
Hashing (Perceptual Simple)	0.08 (0.01)	12753 (190)	0.81 (0.23)	0.28 (0.18)	0.48 (0.13)	0.08 (0.08)	0.36 (0.27)
Euclidean	0.07 (0.01)	11689 (648)	0.84 (0.28)	0.21 (0.18)	0.46 (0.08)	0.001 (0.0)	0.39 (0.36)

Figure 6: Showcasing sample *Maze* levels for different novelty distance functions.

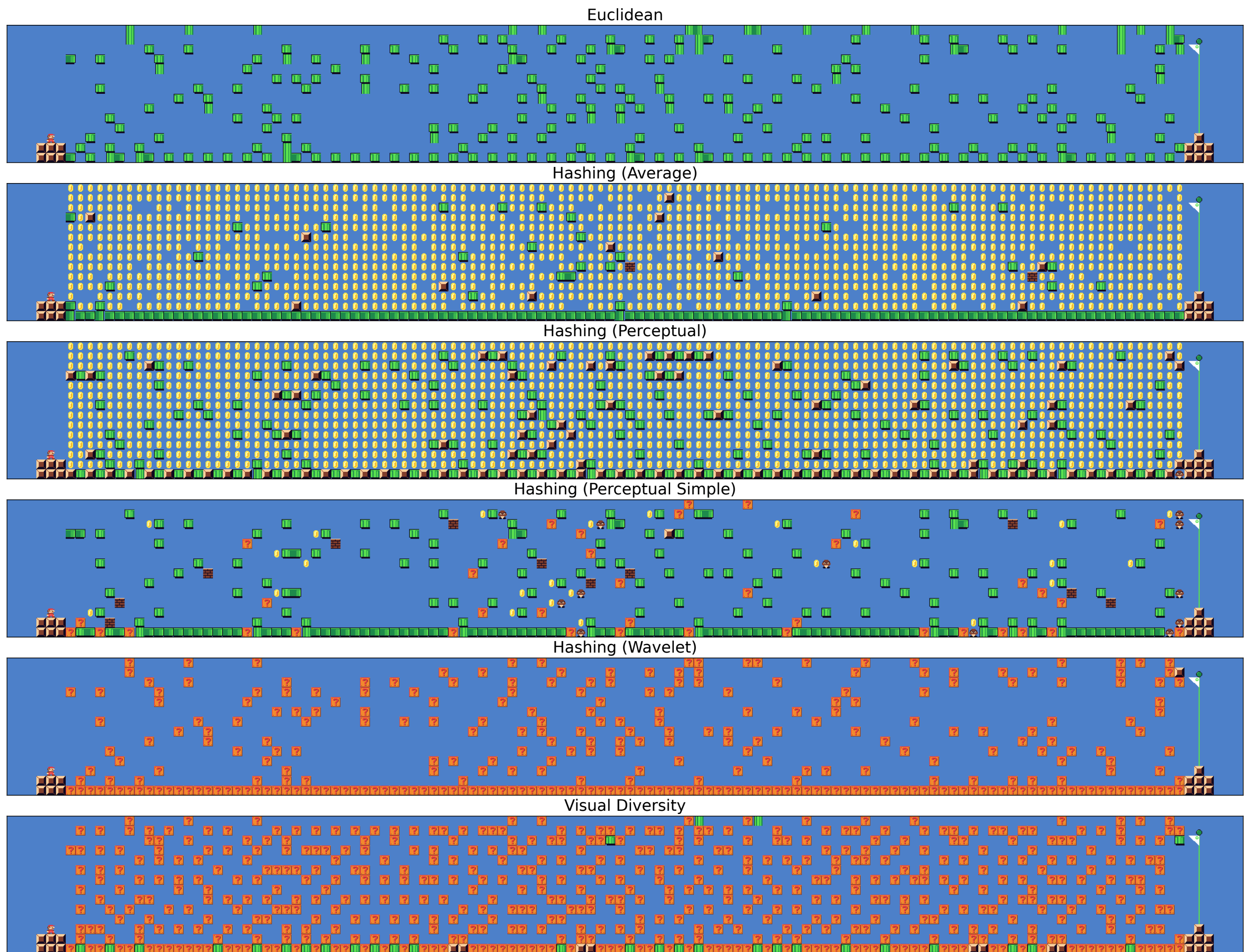
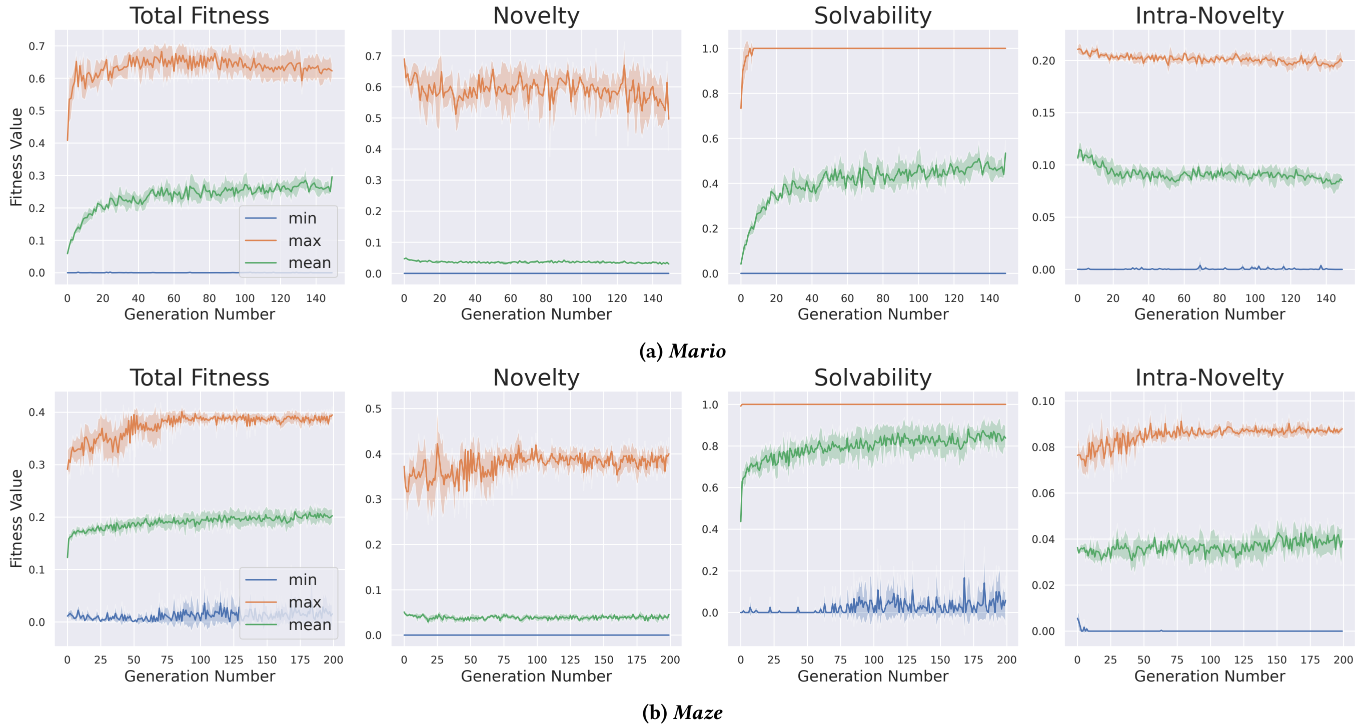


Figure 7: Showcasing sample levels for different novelty distance functions for *Mario*.



(a) *Mario*

(b) *Maze*

Figure 8: Fitness as the training (evolution) process progresses. Standard deviation over five seeds is shaded. For each plot we show the fitness of the best individual (orange), the worst individual (blue), and the average over the entire population (green). Note that the novelty fitness function is relative to the population, and that the intra-novelty fitness is similarly relative to each individual. So, the same novelty fitness at two different generations may not be equivalent, as the populations differ between these generations.