# Procedural Content Generation using Novelty Search for Diverse Video Game Levels

Michael Beukman, Steven James and Christopher Cleghorn
*School of Computer Science and Applied Mathematics*
*University of the Witwatersrand, Johannesburg*

***Abstract*—Procedurally generated video game content has the potential to drastically reduce the content creation budget of game developers and large studios, but many commercial games do not use the power of procedural content generation (PCG) at all. This is because of limitations, like slow generation as well as low quality and diversity of content. We introduce an evolutionary search based approach of evolving level generators using novelty search to procedurally generate diverse levels in real time, without requiring training data or detailed domain knowledge. Evaluating generated levels is often difficult due to the lack of standardised, game-independent metrics, thus we also introduce two simulation-based evaluation metrics that involve analysing the behaviour of an A* agent to measure the diversity and difficulty of a level in a game-independent, general fashion. Our results show an order of magnitude speedup in generation time compared to our baselines, as well as the ability to generalise to arbitrary sized levels without retraining. Our diversity metric is more robust to changes in level size and representation than current methods and additionally measures factors that directly affect playability instead of focusing on visual information.**

***Index Terms*—NEAT, Novelty Search, Procedural Content Generation, Metrics.**

## I. Introduction

Video games are a massive industry, with 214 million reported video game players in the United States as of 2020 [1] and one of the main goals of many video games is to keep players entertained, engaged, and challenged [2]. One way to drive engagement is to have a large amount of unique and interesting content [3]. Most commercial video games still rely on human designers and developers to create this large amount of content [3], but this is costly, and due to tight deadlines, there is always a finite amount of content that players can exhaust relatively quickly.

This concern can be addressed by using procedural content generation (PCG), which refers to “creating game content automatically, through algorithmic means” [4].

There are many notable games in which content and levels are procedurally generated, like *Rogue*, where the player controls a character that traverses procedurally generated dungeons while fighting enemies. The developers only needed to create the mechanics of the game, as well as the method that generates these levels, and now there are a near-infinite amount of levels that players can explore. *No Man’s sky* is also a popular recent game that relied on PCG to create its vast world.

PCG, in general, enables game developers to create engaging and fun games at a fraction of the cost compared to traditional, manual development [3].

There are many different ways of developing a PCG algorithm, from ad hoc algorithmic means like in *Rogue*, to using formal languages [5], to evolutionary search based methods [4, 6], to exhaustive search [7], to more recent approaches of machine learning and deep learning [8, 9, 10].

Different methods have different goals, among others providing a consistent quality of levels, quickly generating playable levels [11], generating diverse levels [12], being configurable [13], etc.

There are gaps in the current literature, however. For example, some methods generate diverse, playable levels, but they are slow [12, 14]. Other methods can generate levels relatively fast, but they require existing training data [15, 16] or game-specific reward engineering [11]. Other methods are limited by the lack of diversity of generated content [8, 17]. 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 the slow generation time by using a trained neural network that can quickly be queried to generate levels. We avoid the need for training data by using an evolutionary algorithm - NeuroEvolution of Augmenting Topologies [18] - to evolve this network. To obtain diverse levels, we explicitly reward diversity in the evolutionary process by using novelty search [19]. This method assigns fitnesses based on how far an individual is from its closest neighbours, to incentivise exploration and obtain diverse behaviours. Additionally, we only use general fitness functions that can be widely applied to evolve these networks - novelty as mentioned above and solvability.

The second problem we address is the lack of comparable metrics between different games and works of literature. For example, leniency [20] measures the difficulty of levels based on atomic pieces of challenge (e.g. a gap in a platformer), which need to be defined separately for each specific game [21] and might not apply to games without a concept of danger to the player, like puzzle games. We introduce two simulation-based metrics that only use the behaviour of an A* agent on a level to judge its difficulty and diversity. Difficulty is measured based on how much the agent needs to explore before solving the level, and the diversity between two levels is calculated by comparing the action trajectories obtained from each level. These metrics require no game-specific feature engineering or domain knowledge, simply a game engine and an agent.

We test our methods on a simple Maze game as well as Super Mario Bros, and our results indicate that our generation method generates levels much faster compared to directly searching for a level or a reinforcement learning-based approach, while mostly obtaining comparable metric scores, and using no game-specific knowledge. We also show that this method generalises to different sized levels than what it was trained on, while still mostly generating solvable levels. Our diversity metric displays some desirable characteristics, like being invariant to visual changes that do not affect how the level is played, and being less sensitive to the size of tested levels than the metrics we compare against.

## II. Background and Related Work

The challenge of procedurally generating video game levels that players find exciting and fun is far from solved. A common problem with generated levels is that they can seem generic and that they do not capture a player's attention and interest them as much as a level that was designed by a developer/designer [6]. Having access to high-quality level generators would help game developers (especially small-scale ones) create entertaining games, with a fraction of the effort and cost [3, 22]. Procedural content generation (PCG) can also be used as a way to train reinforcement learning agents [23], and being able to generate more diverse levels could positively impact the generalisability of these agents.

The research question considered in this report is how to combine novelty search and NeuroEvolution of Augmenting Topologies (NEAT) to procedurally generate diverse and high-quality levels for video games in real time. A secondary question is how to objectively evaluate generated game levels in a game-independent way.

Using the above questions as guiding concepts, we discuss the most applicable previous work on the topic of procedural level generation. We attempt to show that the problem of generating diverse levels in real time has not yet been adequately solved. Finally, we also attempt to show that how levels are evaluated and results are reported leaves something to be desired in terms of an objective, robust, and game-independent metric.

This section is structured based on different aspects of procedurally generating game content. The focus is largely on evolutionary methods, but other approaches are also discussed where appropriate. Section II-A discusses some background related to genetic algorithms and the specific methods we use. Section II-B goes over how content is represented and what space the different learning methods can search in. In Section II-C we go on to discuss the different learning methods that are used and finally in Section II-D we discuss how the content is evaluated and how training signals are generated.

### A. Background

Many of the methods discussed below use some aspects of evolutionary computing, which is a subclass of optimisation algorithms that attempt to mimic natural selection, as found in the natural world [24].

Usually, 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, which can be thought of as 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 [13]. The phenotype, then, is the actual level that has been generated using this specific genotype [24].

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 mutated slightly 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 function could simply be the function value. Using an example from Goldberg [24], if our genotype is a 5 bit binary string and we have parents $A_1 = 01101$, $A_2 = 11000$, we can perform crossover by randomly choosing a number $k \in [1, l-1]$ where l is the length of the strings (5 in this case). We can create two new individuals by swapping the values of $A_1$ and $A_2$ from index $k$ until the end of the string.

If $k = 3$, say, then this would result in two new individuals $A_1' = 01100$, $A_2' = 11001$. Mutation would simply flip bits with some small probability, resulting in (for example) $A_1^m = 01000$ (bit 3 flipped), $A_2^m = 11001$ (no change).
Now, say our problem was maximising the function value $f(x) = x^2$ for $x \in \mathbb{Z}, 0 \leq x \leq 31$. Then our representation (i.e. genotype) could be a 5 digit binary string (e.g. 01100) that represents $x$. The phenotype would then just be the number in decimal (e.g. 12) and the fitness will just be $f(x)$, (which is $12^2 = 144$).

Here we show the above example in table form.

$$\begin{array}{ccccc}
A_1 = 011\mathbf{01} & \overset{crossover}{\Longrightarrow} & A_1' = 011\mathbf{00} & \overset{mutation}{\Longrightarrow} & A_1^m = 01\mathbf{0}00 \\
\uparrow\!\downarrow & & & & \\
A_2 = 110\mathbf{00} & \overset{crossover}{\Longrightarrow} & A_2' = 110\mathbf{01} & \overset{mutation}{\Longrightarrow} & A_2^m = 11001
\end{array}$$

| Genotype | Phenotype | Fitness |
|---|---|---|
| 01100 | 12 | 144 |
| 11001 | 25 | 625 |

*1) NeuroEvolution of Augmenting Topologies :* NeuroEvolution of Augmenting Topologies, referred to as NEAT, is a method where a genetic algorithm optimises the structure and weights of a neural network [18]. This method functions just like a normal genetic algorithm, except that the genotype is the neural network's weights and structure. This method enables complexity to gradually increase as the search process develops, leading to later generations being more complex than previous ones.

*2) Novelty Search:* Novelty search [19] is a different way of designing the fitness function for 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 way, e.g. using a vector norm, like absolute difference in our above example or Euclidean distance in a higher-dimensional space. 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 [25, 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 lead to agents being stuck in local minima [19].

### *B. Level Representation*

How a piece of content is represented in a specific method can impact the type of levels that method can generate, its efficiency as well as the ease of crossover and mutation when using a genetic algorithm.

Togelius et al. [4] make a distinction between indirect encodings and direct encodings, where the dividing line is whether the representation size is proportional to the level size. This means that a direct encoding for a level of twice the size requires a representation that is twice as large. Ferreira et al. [13] used a simple, direct vector representation where the size of each vector was the same as the length of the map (each aspect of the map, e.g. enemies, coins, and ground used a different vector). To generate a map from this representation, one simply iterates over the vector and places the appropriate component at that spot. This is a direct encoding, where it is very easy to map between the representation and the level. Liapis et al. [12] used a 2D array of tile types to represent a 2D grid world. This facilitated easy crossover, but it occasionally caused unplayable levels. The authors remedied some of these problems by using a simple, game dependant repair mechanism that removed some tiles if there were too many of a specific type.

Risi et al. [27] used a more indirect representation, where each collectible game item (flower) was represented by a compositional pattern producing network (CPPN) [28], which is a type of artificial neural network where each node can use a different activation function. This network could generate a single item by outputting an RGB colour value for each given polar coordinate. Thus, the representation was actually the structure and weights of this network. This method has the potential to scale up, where larger items (more pixels or more layers) can be generated using the same size representation. This can also be viewed as a form of content compression.

One potential downside of indirect encodings is that there might exist some area in the content space that cannot be generated by any genotype [4]. This problem is relatively easy to spot when using a direct representation.

*1) Search Space:* The representation determines the search space of the learning method. There are two common approaches.

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 encoding here is usually quite direct. Examples of this include the work by Ferreira et al. [13], where the method searched over integer vectors of the same length as the level (200), Liapis et al. [12, 29] searched over a 2D tilemap representation of the levels and Cardamone et al. [30] generated 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 [11] or the weights of a neural network. Kerssemakers et al. [31] used a genetic algorithm to search for parameter vectors that determine the behaviour of a non-deterministic (and thus reusable) generator.

Risi et al. [27] used a method that is somewhere in between. The method evolves a single neural network for each game item (which is an image of a flower). Different parts (layers) of the item are generated using a different layer input to the network and these are then stacked on top of each other to form the final item. This is actually still searching in 'level' space, however, since each representation generates only a single item, even though the representation is indirect. Potentially, though the authors did not explore this avenue, each of these networks can generate multiple items, by using sets of disjoint layer numbers (e.g. item 1 uses layers 1-5, item 2 uses layer numbers 6-10, etc.).

The main advantage of using a direct representation and searching in level space is that it is quite simple to design a genotype and genotype to phenotype mapping (in our example, we simply converted the binary string to a decimal number). This conversion usually does not require much computation, e.g. Ferreira et al. [13] simply iterate over the vector representation of the level and place the item at that index at the corresponding position in the level. This method can suffer performance problems, however, because each new level necessitates performing a whole new search, which does take time. These methods are therefore unsuited to real-time, online level generation, although there are some exceptions, like Galactic Arms Race [32] which evolves weapons in real time. Togelius et al. [4] describe another potential issue when using search-based methods, that of "Catastrophic Failure" [33], which can result in either unplayable or low-quality content or long search times to make up for a bad random initialisation. This risk might be unacceptable for certain commercial games.

Searching in generator space is more challenging, as one has to design some mapping that the generator can make between a parameter vector (or random seed) and a game level. But, since a generator is learned, one can move the computational burden to an offline training stage and still achieve real-time level generation by simply querying a policy, which makes it suitable for generating levels on the fly in games. Khalifa et al. [11] frame the level generation task as a sequential process, where at each step the agent receives some observation (i.e. the whole map and some extra information) and performs a single action, which could be to change the current tile or a specific tile of its choosing. This approach is very fast and does not require prior training data. It does have a possible limitation, though, in that it assumes that the level generation process is sequential, which might not be appropriate for some games levels, where a more global design sense is desired. A second limitation is that the training required is substantial, on the order of 100 million timesteps, which might limit experimentation.

Kerssemakers et al. [31] use a two-phased approach to create a level generator. The inner generator consists of a collection of non-deterministic agents that move through the level while modifying it. These agents are parametrised by a single vector, which is evolved using the outer generator. Since the agents are not deterministic, each inner generator can generate a large number of levels. The authors showed that this method can generate levels in real time and that, depending on the specific generator used, it can also generate diverse levels. Since the agents cannot communicate with each other and some agents are only active in certain parts of the levels, the variation between different parts of a level can be quite large, which might lead to some perceived dissonance on the player's part.

In Table 1 we summarise some of the advantages and disadvantages of each method.

| Generator Space | Level Space |
|---|---|
| Fast generation | Slow generation |
| Difficult to design | Simpler to design |
| Easier to generalise | Game-specific |
| Long training time | No training time |
| Relatively consistent quality | Can have large variance in quality |

Table I: A comparison of searching in generator space vs searching in level space.

### C. Learning Methods

Historically, evolutionary methods have been some of the most popular approaches to PCG [4, 8]. These methods are usually simple to implement and one does not need a labelled set of training data to be able to generate levels.

For example, Ferreira et al. [13] used a simple genetic algorithm, whereas Liapis et al. [12] used a 2 population genetic algorithm with the novelty metric as the fitness function. Using two populations - one consisting of feasible individuals and the other of infeasible individuals - was used to ensure a high number of playable levels.

Liapis et al. [12] discussed two different methods that can be used depending on the goal - generating diverse levels or generating playable levels relatively quickly. Due to the high computational costs involved, however, these methods are not always suitable for real-time level generation.

Recently, however, more focus has been put on applying machine learning/deep learning to this task [8, 9]. Khalifa et al. [11] used reinforcement learning, by formulating the task of level generation as a Markov Decision Process (MDP) and using existing methods. The agent generated the levels sequentially by performing a specific action (which involved changing a single tile of a 2D map) at each time step.

There has also been work on using a combination of the above methods, for example, Risi et al. [27] used a neural network to represent collectible items, but evolved the structure and weights using NEAT [18]. This method is still largely evolutionary, however, since there is no other learning mechanism in play.

Another notable work is the one by Volz et al. [16] where the authors used unsupervised learning to train a generative adversarial network (GAN) on a collection of existing game levels. They then used evolutionary methods to search for a latent input vector to this GAN, which generates the level. Building up from this, Schrum et al. [15] went one step further, by instead evolving a CPPN [28] that takes in the coordinates of a level segment and outputs the latent vector that the GAN uses to generate that segment. Many of these segments are then combined to form the complete level. This method has the advantage of being able to scale up to larger levels with the same CPPN and GAN, as one would just use different coordinates for later segments.

The main disadvantage with the more traditional machine learning-based approaches is that existing training data is required, which limits its usefulness for creating levels for new games. Another subtle effect of this need for data is that it can influence the model to generate levels that are close in character to the training data. This might be desirable, but there could also be cases where a user would like to create new and innovative levels using a PCG system.

### D. Evaluation

The training/reward/fitness signals used in these methods to evaluate a piece of content are very important. Using the wrong learning signal can result in the learning or searching process not resulting in desired levels [4, 34].

One way to evaluate content is to let users pick those pieces of content that they prefer, which is referred to as interactive evaluation [4].

This means that a human rating/score is the fitness function, and there is not an explicitly defined objective (as we had in the simple example above). Cardamone et al. [30] generate racing game tracks and let users rate the generated levels, which serves as the fitness function. Risi et al. [27] used the same structure, where users can breed the specific collectible items that they like, thus guiding the evolutionary process. This method does not have an explicit measure of fitness though, as users are able to manually choose which parents to use for crossover (as opposed to ranking individuals and letting the algorithm choose the specific combinations). Kerssemakers et al. [31] used a similar, interactive method to evolve level generators. To facilitate finding payable levels early on, the evolution was seeded with a set of highly playable levels that were generated offline and without interactivity. The authors do state that if a user wants to generate levels with a very specific feature, it might still take a long time for the population to evolve towards that goal. Another potential downside of this method of evaluation is that the population size one can use is severely limited, which might impact how much of the space the population can search and how effective the searching is [35].

This type of fitness function is useful in cases where a synthetic one is difficult to design, or where it is desirable to have humans guiding the process. The downside is that it is asynchronous and slow, as one has to wait for humans to evaluate content before continuing with the search. It can also interrupt the player's experience. This is fine in a game like *Petalz* [27], where the whole game revolves around this mechanic, but might not be appropriate when fast and automatic level generation is desired. One can also use implicit data collection, where the game tracks player action information in the background and uses that as the metric by which to evaluate the level. This data, however, can often be difficult to collect, noisy and not of much use [4].

Other methods, called direct evaluation, are largely similar in that they involve the designer creating some notion of fitness that can easily be computed. Ferreira et al. [13] used simple rules to evaluate Super Mario Bros levels, like using the sparsity of the enemies and the entropy of the ground. This does require some domain knowledge and it can be difficult to design a metric that guides learning successfully towards desired goals. Khalifa et al. [11], who used reinforcement learning to incrementally change a level, created game-specific modules that assessed the difference in quality each incremental change results in and used that as the reward signal.

Another approach in a similar vein is to use a more high-level fitness function that measures the challenge and expected enjoyment of a level and does not focus on the low-level features [36]. This type of method can be a useful way to generalise fitness functions across games but might be liable to guiding the learning method away from the intended result and towards undesirable results that maximise this more indirect metric.

A recent work that optimised a higher level reward function, specifically maximising notions of fun and diversity was EDRL [10], which combined the above idea of PCGRL [11] and experience-driven PCG [37] to generate potentially endless levels by making use of a GAN based chunk generator, and using reinforcement learning to choose the latent input vectors to this GAN.

Domain independent methods that do not require as much game-specific designing can also be used, which enables easier reuse and transfer to other games.
Liapis et al. [12] used the notion of the *novelty metric* [19], which was described in the background section, to evaluate feasible levels and used either the novelty metric or the distance to feasibility for infeasible levels. The distance function was defined as the visual difference between two tilemaps. This approach leads to the individuals in the population having behaviours that are diverse and novel. It does forsake a general objective though, which can cause the algorithm to focus on exploring the vast infeasible space. This is why the authors used a 2 population (feasible and infeasible levels) algorithm to protect the feasible individuals from being killed off for not being novel enough. This can result in a high level of diversity between the levels, as well as levels that have desirable qualities. This level of diversity is usually difficult to obtain when using standard evolutionary methods.

Another way to evaluate a generated level is using an artificial agent that plays the level [4] and deriving the score based on how the agent played (e.g. how quickly the agent solved the level, its total distance travelled, etc.). One can even let the agent learn when playing the level and possibly incorporate that into the evaluation function.

*1) Constraints:* Another aspect of evaluation is how to deal with infeasible individuals that do not satisfy some of the many constraints (e.g. there can only be one player, the level must be solvable, etc.). There are different methods to address these concerns, for example, Liapis et al. [12, 29] use two different populations, where one consists of the infeasible individuals and the other contains the feasible ones. Offspring from either population can migrate to the other if they satisfy the feasibility/infeasibility condition. They independently evolve these two populations, where individuals only compete within their own population. This results in a larger number of feasible levels because these individuals are not immediately killed off for not being as novel as the infeasible individuals.

A constraint satisfaction method can also be used to repair unplayable levels that resulted from crossover or mutation [36]. In general, however, to design such a system can require a significant amount of domain knowledge and might even be as difficult as creating an explicit level designing algorithm [12].

Other methods just kill off the individual by setting the fitness/reward to 0 when encountering an infeasible individual [38, 39]. This can result in a low amount of diversity, or low-quality levels, because the best levels are often close to the border between feasibility and infeasibility [29, 40].

*2) Final Evaluation:* How one chooses to evaluate the generated levels for comparison between different methods is also important. For example, Khalifa et al. [11] used some preset 'goal' criteria to evaluate the level. These criteria stipulated that the level has to be solvable in more than X steps, it should have only 1 player, 1 door and 1 key, etc. Liapis et al. [12] used several different metrics, namely how many runs resulted in a feasible individual being found, the final number of feasible individuals and their average diversity, which is measured as the average number of non-matching tiles in a tilemap. Ferreira et al. [13] simply state that their method can generate levels that are similar to the original Super Mario Bros levels.

It is quite difficult to compare these different results, as they usually measure different aspects, and often do not use the same units or methods of measuring.

It would be useful to have a consistent way of evaluating a level that potentially includes metrics such as diversity, difficulty, and quality. Horn et al. [41] echo this sentiment and attempt to create a valid range of metrics that can be used to evaluate levels, but these have not been widely adopted. Most of the metrics are defined for a single level, measuring aspects like linearity, density, and leniency (how easy a level is). The one metric that compares the diversity of levels is the compression distance, which measures the relative similarity between two levels from the same generator. The authors add that promising future work could be to extend these metrics by adding in a simulation-based evaluation score, where an artificial agent plays the generated level, and the score is derived from the playing style of this agent.

### *E. Conclusion*

Even though there is a lot of work in the PCG field, there are still some unsolved problems. The main one is being able

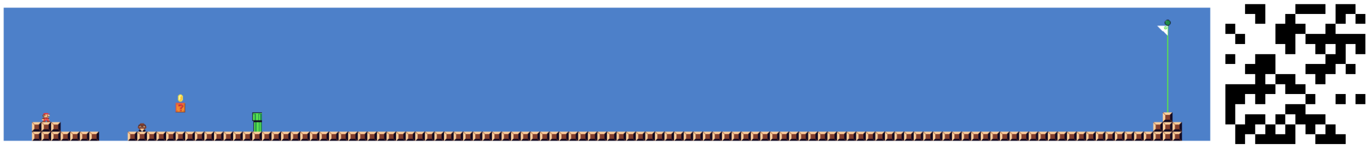


Fig. 1: Example levels - Mario and Maze.

to generate (1) high quality, diverse levels that (2) can be generated in real time by (3) a learned generator (4) without existing training data.

A second shortcoming is the evaluation metrics used when reporting results. These methods often prohibit comparison between different works and they can be very game-specific, which makes it difficult to compare methods across different games. These metrics are also not very robust to what humans see as novel or diverse. A simulation-based, game-independent metric will enable sensibly comparing different generators.

## III. Measuring the characteristics of levels using A* agents

Here we describe how our metrics of diversity and difficulty function, and how exactly we use the A* agents. These agent-based metrics are quite general and evaluate the diversity and difficulty of generated levels without requiring game-specific configuration or human interaction. We compare them to the compression distance and leniency metrics, respectively.

### A. Games

For this entire report, we investigate two different tilemap-based games. The first is a simple Maze game where each tile can be a wall or empty space, and the goal is to find a path between the top left corner and the bottom right corner. The second game we consider is a simplified version of Super Mario Bros, with only Goombas as enemies, and no powerups. Examples can be seen in Fig. 1. The Maze game acts as a simple proof of concept, and Super Mario Bros is widely used to test procedural level generation systems [10, 13, 21, 36].

### B. Existing Metrics

Here we list two of the existing metrics that we compare against.

*1) Compression Distance:* Compression distance (CD) is a metric that measures the similarity between two strings by determining how much space is spared by compressing them as one concatenated string, compared to compressing them separately [42].

The *normalised compression distance* (NCD) is defined as

$$\text{NCD}(x, y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$

where $C(x)$ is the length of the string $x$ after compressing it, and $xy$ is simply the concatenation of $x$ and $y$. The intuition is that strings that are very similar will have a low NCD value as compressing them together can take advantage of their similarities. On the other hand, strings that are very different will have a higher distance, as compressing them as one string does not offer much benefit over compressing them separately. For clarity, we refer to this metric (using gzip as the compression algorithm) simply as compression distance (CD).

Shaker et al. [21] applied this to platformer levels by generating a feature for every column in the map, which is determined by different aspects, like an increase or decrease in platform height, the start or end of a gap, the existence of enemies and other blocks, as well as their combinations. This string of features is then used as the string representation of a level.

For the Maze Game, we simply flatten the map, and thereby obtain a binary string that represents the level.

*2) Leniency:* Leniency is a metric that measures how lenient a level is to a player's mistakes. Originally defined for platformers by Smith et al. [20], it was determined by averaging the *leniency score* for each of the potential challenges in a level, and normalising that to a value between 0 and 1. These challenges could be a gap or an enemy (with a leniency value of $l = -1$), a jump with no associated gap ($l = 1$), and various other elements. We follow the approach of Shaker et al. [21], who used leniency in the context of *Super Mario Bros* by slightly adapting the metric described above.

For the Maze game, however, since there are no inherent challenges, we calculate leniency as the fraction of dead ends there are in the level. For each unfilled tile $t$, we find the shortest path from the start $s$ to $t$. We then fill in this path and determine if there is still a path from $t$ to the goal $g$. If there is, then $t$ is not labelled as a dead end, otherwise it is.

### C. A* Diversity

For diversity, we assume that diverse levels require diverse solutions [43], and we thus look at the differences in solution characteristics between two levels to calculate their diversity.

To measure the diversity between two levels, we simply run our A* agent separately on each level, and consider the actions that the agent performed, as integers specifying a single discrete action (like moving to the right in the Maze game). We use these action strings to evaluate how different the levels are, using the Levenshtein (or edit) distance [44], and normalise this by dividing by the length of the longer string. This has the added benefit of being able to compare trajectories that are not necessarily the same length.

This diversity metric measures levels that can be solved using similar approaches as similar, and other levels that have drastically different paths as diverse. This method thus focuses on the characteristics of a level that actually affect the playability, instead of being distracted by different (but potentially irrelevant) visual information.

There might be some potential problems with this metric, however. Specifically, we use actions to distinguish between levels, and some actions might not affect the environment, like walking into a wall [43]. This is mitigated by using an agent that rarely performs unnecessary actions. A second possible shortcoming is that multiple actions might result in roughly the same path, (e.g. in Mario, jumping to the right vs just moving to the right), leading to noise in the metric. Again, using the same agent in all cases leads to consistent and comparable actions. Secondly, we compute the correlation between this edit distance-based metric and a similar one that uses the average Manhattan distance between corresponding positions (which should be less affected by small differences in actions), and we found a high correlation (Pearson's $r > 0.7, p < 0.05$), indicating that this phenomenon does not have a very large effect.

Note, this method is not constrained to only use A*, and it can be used with any game-playing agent, like an evolutionary controller or reinforcement learning agent.

### *D. A* Difficulty*

We measure the difficulty of a level as the number of nodes that the A* agent expanded that are not along the optimal path divided by the total number of possible states that could have been expanded. The justification for this is that levels that are more difficult and contain more dead-ends will require more exploration, and thus more expansion of the search tree. Easier levels on the other hand, where there is a simple, unobstructed path between the start and the goal will require much less searching, as the heuristic will lead the agent directly to the goal without much searching or exploration.

### *E. Results*

*1) Diversity:* When comparing these two diversity metrics, we found a few specific characteristics and limitations of compression distance:

1) Compression distance is quite sensitive to the exact string representation used, and there is a marked difference between simply flattening the Mario levels and using a more game-specific feature representation.
2) As the level size increases, compression distance (a) increases as well and (b) the variance decreases significantly. Larger levels are thus all marked as having more or less the same diversity score.

For the following plots, we consider the distribution of diversity scores, i.e. if we have 100 levels, there are $\frac{(100)(99)}{2} = 4950$ pairwise comparisons, and we consider all of these values for 5 different seeds.

Fig. 2 shows that the compression distance is quite sensitive to the exact representation used. We consider three different representations, firstly 'Normal', as described above and detailed by [21, 45]. Secondly, we use 'Concatenated', which is similar to Normal, but we instead concatenate the strings representing the platform height and the enemy placement, respectively [45]. Finally, we consider 'Flat', where we simply flatten the 2D array of tiles. Table II shows that the correlation between these methods, especially for the Flat representation, while positive, is not very large. Normal and Concatenated do seem to be better correlated, however, as they are related.

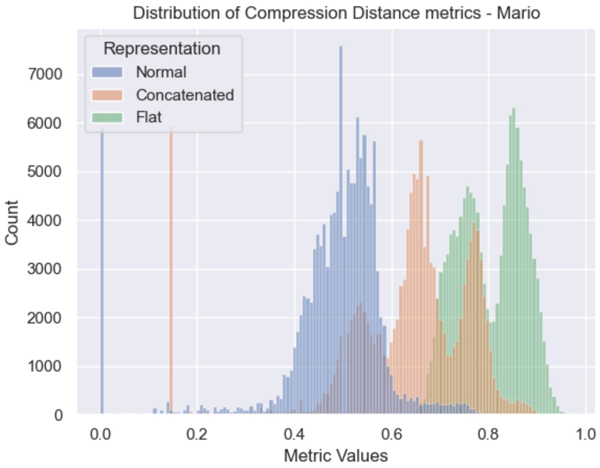


Fig. 2: Mario - Distributions of compression distance when using 3 different string representations.

| | Normal | Concatenated | Flat |
|---|---|---|---|
| Normal | 1.000000 | 0.838985 | 0.129255 |
| Concatenated | 0.838985 | 1.000000 | 0.329778 |
| Flat | 0.129255 | 0.329778 | 1.000000 |

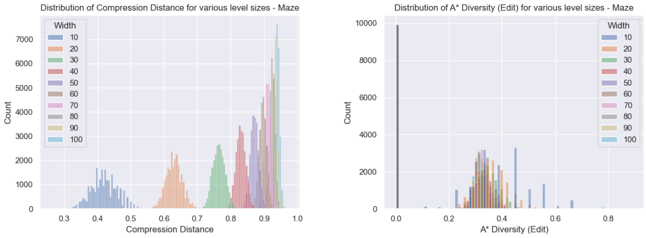
Table II: Pearson's correlation coefficient between different compression distance representations - all had a statistically significant $p < 0.05$. More than 120 000 pairwise comparisons were used to calculate the correlation.

Fig. 3 shows that as the level size increases, the compression distance's mean becomes larger, and it varies less. In contrast, the A* diversity metric shows less dramatic changes when the level size increases. Another interesting point to note is that our metric marks many pairs of levels as being identical (i.e. diversity $= 0$), as the same set of actions solved both levels, which is something that the compression distance does not do. We see a similar effect for Mario in Fig. 4, but it is much more pronounced when using the flat representation.

Thus, compression distance has some undesirable characteristics that make it less suitable as a comparable, objective metric. Concretely, it is very dependent on the specific string representation used, as well as the size of the levels under consideration. The A* diversity metric is less affected by these changes.

*2) Difficulty:* Here we perform some analysis of the A* difficulty metric, and how it compares to leniency.

When comparing the leniency of a collection of levels with their A* diversity score, we find a Pearson's correlation coefficient of $-0.17$ ($p = 3.4 \times 10^{-30}$) over about 4000 levels for the Maze, and we find no statistically significant correlation over about 2500 levels for Mario. This might



(a) Compression Distance

(b) A* Diversity

Fig. 3: Maze - Distributions of metrics. Each colour represents the metric values from pairwise comparisons over 5 seeds and 100 levels per seed for different sized levels. Only solvable levels that were generated using one specific method were considered (to fairly compare against the A* metric, which requires solvability), but the compression distance trend holds for random levels as well.

indicate that Leniency and A* difficulty measure different aspects of challenge.

We next investigate whether there exists a correlation between the original Super Mario Bros levels and the metrics above, specifically, whether later levels imply a higher difficulty (lower leniency) than earlier levels. We use the implementation[1] from Horn et al. [41] to measure the leniency of a subset of 10 of the original levels and we find no statistically significant correlation between the level index and difficulty for either leniency or the A* metric. During this experiment, we also found that some levels were not solvable by the A* agent, since it usually took a greedy path to get to the rightmost point of the screen, and some levels had barriers that needed to be surpassed more cleverly.

### F. Conclusion

This section introduced diversity and difficulty metrics that both depend only on a game-playing agent. We compared these to existing approaches and found that the diversity metric does not have some of the same problems as compression distance - being dependent on visual changes, level size and string representation. Our difficulty metric does not correlate very well with the existing leniency metric.

## IV. Generating Diverse Levels Quickly

Here we detail our method, Procedural Content Generation using NEAT and novelty search (PCGNN) that generates diverse levels, in real time, without any existing training data or game-specific learning signals.

As previously mentioned, we use NEAT [18] to evolve a neural network that generates levels. This is split up into two parts, training (evolution) and inference (generating levels).

The next few sections are dedicated to detailing these components, focusing on how levels are generated in Section IV-A, discussing the training process in Section IV-B and describing the fitness functions we use in Section IV-C.

### A. 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 red one in Fig. 5), we input the surrounding tiles (the pink tiles in Fig. 5) into the network, using either integer tile values or one-hot encoding, and use the output to predict what tile type should be placed at the original location. To predict the boundary tiles, we pad the level with $-1$. 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 [10, 15, 16], and make locally consistent choices [8, 46]. We input some 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 [47].

This process is performed sequentially, so the previous predictions become part of the level, and thus the network will receive them as inputs when predicting the next tiles.

This method of generation also provides some configuration options, specifically how big the context size should be (i.e. the pink region in Fig. 5), so we could for example use the 3 adjacent rows and columns instead of just the closest ones. The other option we have is to increase the number of tiles predicted at each step, so instead of one as in Fig. 5, we could predict a $2 \times 2$ region, in effect providing a middle ground between generating only one tile and generating the entire level at once.

### B. Training Process

The training process is as follows. We use the standard NEAT algorithm [18], and evolve the population as normal, 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 could also be adapted, however, and multiple level generators could be obtained, with each potentially having a unique style. This final generator can then be used to rapidly generate many new levels, as generation only requires the network to be queried, and no learning or searching is done in this step.

### C. Fitness Functions

To evaluate a network, we first generate $N$ levels and use these to compute the fitnesses. The fitness of the network is then simply the average fitness of its levels. We use different fitness functions for different purposes, and unless otherwise stated, all fitnesses are scaled to between 0 and 1. When using multiple fitness functions, we simply set the final objective as a weighted sum of these fitnesses, as this was an easy way to combine them, and Gomes et al. [48] found that this method performs comparably to using multi-objective optimisation techniques specifically when using the novelty metric.

**Solvability** This simply determines if the level is solvable, using a simple graph algorithm like A*. The fitness value is 1

[1] http://sokath.com/fdg2014_pcg_evaluation/

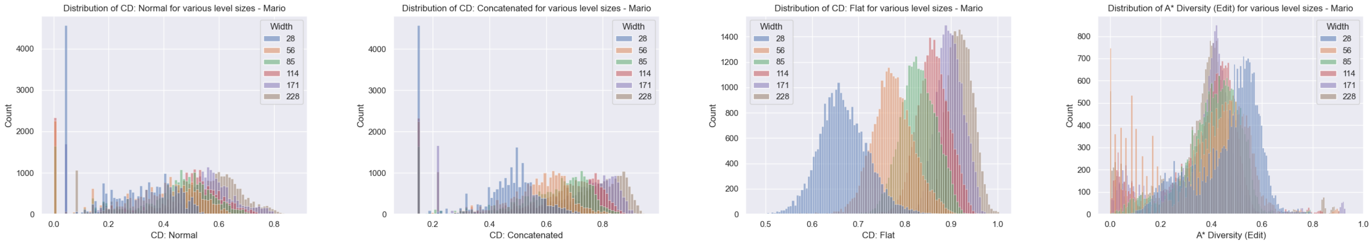
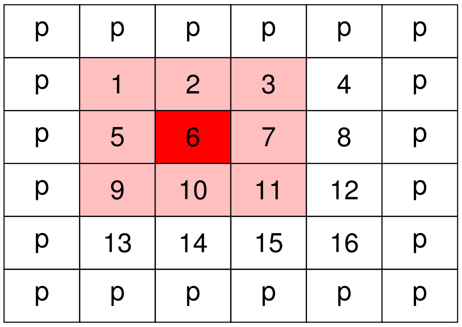


Fig. 4: Mario - Diversity metrics as level size increases.



| | | | | | |
|---|---|---|---|---|---|
| p | p | p | p | p | p |
| p | 1 | 2 | 3 | 4 | p |
| p | 5 | 6 | 7 | 8 | p |
| p | 9 | 10 | 11 | 12 | p |
| p | 13 | 14 | 15 | 16 | p |
| p | p | p | p | p | p |

Fig. 5: An illustration of the level generation process. p indicates padding.

if the level is solvable and 0 otherwise.

**Novelty** We also use the novelty metric [19] 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, among others the Visual Diversity [12], which measures the fraction of non-matching tiles (i.e. the normalised Hamming distance), perceptual image hashing [49, 50], which gives a low value for images that look similar and large values for those that look different, or any other valid distance function. The exact function used can have a large effect on the diversity characteristics of the generated levels [14]. 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 $\lambda$ individuals to it at every generation [48] instead of adding only individuals that obtained a high novelty, as in the original work [19]. This is generally preferable, as it leads to more robustness of parameters and more uniform exploration [48].

**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.

**Other** Other fitness functions can also be used to generate levels with some specific property, like having longer shortest paths, more connected regions [11], and some other feasibility criteria that do not affect solvability, like not placing enemies in mid-air. These could be used to inject some expert knowledge into the generation, or to enforce some constraints. We do not use these in our experiments, however.

Algorithm 1 shows pseudocode for the training procedure.

### D. Conclusion

In 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, 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.

## V. Experiments

Here we detail our experimental setup, what our baselines are, and what metrics we use to evaluate the levels. The next section covers the results we obtained.

### A. Baselines

We use a few baselines to compare our methods against. The first is a genetic algorithm-based approach by Ferreira et al. [13], which was originally developed for Super Mario Bros. 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 the fitness function being either entropy [51] or sparseness [52]. For the Maze, we simply use the 2D grid directly as the genotype, using partial solvability and entropy as the fitness functions.

**Algorithm 1** Novelty NEAT Training Procedure

```
Require: $G \geq 1$ ▷ Number of generations
Require: $N \geq 1$ ▷ Number of levels to generate per network
Require: $K \geq 1$ ▷ Number of neighbours for novelty
  $Pop \leftarrow$ random initial population of networks
  for all generations $g \in \{1, ..., G\}$ do
  Phase 1 – Generate Levels + Calculate Simple Fitnesses
      for all $net_i \in Pop$ do
        Generate $N$ levels, $L_{i1}, ..., L_{iN}$ from network $net_i$
          ▷ Fitness is e.g. Solvability and/or Intra-Novelty
        $Fitness(net_i) \leftarrow mean(Fitness(L_{ij})), \forall j$
      end for
  Phase 2 – Calculate Pairwise Distance
      Set $Distance(i, i) = \infty, \forall i$
      for all $net_i \in Pop$ do
          for all $net_j \in Pop, j > i$ do
              ▷ $d_l$ could be e.g. Visual Diversity
              $Distance(i, j) \leftarrow \frac{1}{N} \sum_{n=1}^{N} d_l(L_{in}, L_{jn})$
              $Distance(j, i) \leftarrow Distance(i, j)$
          end for
      end for
  Phase 3 – Calculate Novelty Fitness
      for all $net_i \in Pop$ do
          Sort $Distance(i, \cdot)$ ascendingly
          $Novelty \leftarrow \frac{1}{K} \sum_{k=1}^{K} Distance(i, k)$
          ▷ Linear combination of different fitnesses
          $Fitness(net_i) \leftarrow w_f \cdot Fitness(net_i) + w_n \cdot Novelty$
      end for
      Update population using calculated fitnesses.
  end for
  Return best individual network.
```

Partial solvability is a less sparse version of the solvability fitness function described previously, directly applicable for the Maze, and it returns $\frac{1}{3}$ for each of: the starting tile is empty, the goal tile is empty, these two are connected. 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 [51] of each chunk, and return the average. Calculating the entropy of a chunk works as follows: For each tile type $t$ (e.g. filled and empty for the Maze game), we count the number of tiles that have that value. From here we can make 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

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

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. [12], 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) [11], where the level generation process is modelled as a reinforcement learning problem. For the Maze, we use reward functions that reward 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 fast generation time.

We do not consider CPPN2GAN [15] or the approach by Volz et al. [16] as baselines, as these methods require existing levels as training data. Similarly, EDRL [10] uses a GAN based chunk generator, which was trained on existing Super Mario Bros levels, though in principle any parametrised generator could work.

### B. Metrics

To determine the characteristics of the levels that we generate, we use some different metrics, which are described here.

*1) Solvability:* Solvability is a relatively simple metric, we simply determine if the level is solvable, using a Breadth-first search or A* agent to traverse the level from the starting point to the ending point.

*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.

*3) Other:* The other metrics we use were described above in Section III, specifically compression distance, leniency, as well as the A* diversity and difficulty metrics.

### C. Implementation details

We use a standard implementation[3] of NEAT to perform the evolutionary process. For the Maze, the network has a single output neuron that is thresholded (by 0.5) to obtain a prediction for that tile (e.g. 1 = full, 0 = empty). For Mario, we one-hot encode the input tiles, and the network has multiple output neurons, one for each tile type. The tile is chosen that corresponds to the neuron with the largest response.

*1) Agents:* For Mario, we use the A* agent by Robin Baumgarten[4], which won the first Mario-AI competition in

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

[3] https://github.com/CodeReclaimers/neat-python

[4] https://github.com/amidos2006/Mario-AI-Framework/tree/master/src/agents/robinBaumgarten

2009 [53, 54]. We also use the updated Mario-AI framework[5] to import and evaluate our generated levels.

Since a level might be solvable in multiple different ways, and the A* agent sometimes performs differently on the same level for the Mario game (due to the relative complexity of the simulation), we run the diversity and difficulty metrics 5 times on each level (for both games) and simply average the results. For the solvability metric in the Mario case, we also run the agent 5 times, and if it solved the level in any of these 5 runs, we label it as solvable.

Our Mario solvability fitness function uses a less complex, but much faster simulation[6], and we only use the full fidelity simulation when evaluating the levels, after training and generation. The Maze game uses a simple implementation of A*.

### *D. Experimental Setup*

We compare our method of level generation against the aforementioned baselines using the metrics discussed above.

For all generation methods, we performed a hyperparameter search and in all cases report the best result obtained.

All level generation experiments were run over 5 different random seeds, and the average and standard deviation are reported here. Each of these 5 runs consisted of generating 100 different levels, and the metrics of the 100 levels were 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. [41] and measure the diversity between a group of $N$ levels by calculating calculate 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.

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

**NoveltyNEAT** This is our NEAT based method that uses novelty search.

**DirectGA** Only applicable to Mario, this is the genetic algorithm with the same parameters as in the original work [13].

**DirectGA+** The DirectGA method (for both Maze and Mario), where we performed 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 / turtle representations, respectively [11].

For PCGRL, we attempt to train the method for the same 100 million timesteps as stated by Khalifa et al. [11], but the Mario training process was slower than the Maze (possibly due to the much larger level size under consideration), 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 single level to 10000, as without this the PCGRL model sometimes became stuck in a loop.

*1) Statistical Approach:* For the following sections, we use the notation $\mu_{\mathrm{b}}(M)$ to refer to the mean of metric $M$ when using method $b$, and $\sigma_{\mathrm{b}}(M)$ denotes the standard deviation. All results in the following tables are of the form $\mu$ $(\sigma)$.

For most of the metrics tested, at least one method failed a Shapiro Wilks normality test [55] with $p < 0.05$. Thus, for consistency, all statistical tests performed below are nonparametric Mann-Whitney U tests [56], either one- or two-sided. For each metric, we separately compare our method's results against each of the baselines. If we find a statistically significant result, we also calculate the Cohen's d value [57], $d = \frac{\mu_{\text{neat}}(M) - \mu_b(M)}{\sigma_{\text{neat}(M)}}$ to measure how large this difference is.

In the tables shown in the next section, we use **bold** to denote a statistically significant result ($p < 0.05$), $^*$ to indicate a $p$ value less than $0.01$, and $^\dagger$ to denote a large effect size ($|d| > 0.8$).

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

*2) Structure:* We compare our method against the baselines on generation time in Section VI-A and on solvability, diversity and difficulty in Sections VI-B, VI-C and VI-D respectively. We finally consider generalisability in Section VI-E.

## VI. Results

### *A. 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

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

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

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

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

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

[5] https://github.com/amidos2006/Mario-AI-Framework

[6] https://github.com/amidos2006/gym-pcgrl/blob/master/gym_pcgrl/envs/probs/smb/engine.py

We thus reject the null hypothesis that our method's generation time is comparable or slower than our baselines. 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), which does make sense, as 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 NoveltyNEAT and DirectGA.

We do not parallelise the fitness function calculations for either DirectGA or NoveltyNEAT, although doing so could improve performance, notably generation time for DirectGA. The reason we do not do this is that this would largely depend on the number of cores the generating machine has, and many cores are often not given on a player's machine [58]. Similarly, we only take the top individual after performing the generations, and it would be more efficient to take more individuals from the final generation, but these individuals might lack in diversity or feasibility [17].

### B. 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 IV shows the results.

| | Maze Solvability | Mario Solvability |
|---|---|---|
| NoveltyNEAT (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)**$^{*\dagger}$ | 1.0 (0) |
| PCGRL (Wide) | 1.0 (0) | **0.74 (0.03)**$^{\dagger}$ |
| PCGRL (Turtle) | 0.95 (0.08) | **0.70 (0.02)**$^{\dagger}$ |

Table IV: Solvability. Higher is better.

We find no statistically significant difference between our solvability, the solvability of PCGRL and that of DirectGA+ for the Maze domain. For Mario, we are comparable to all DirectGA implementations, and significantly different from both PCGRL representations. 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 on the Maze domain, and still very high on the Mario domain.

The above result, coupled with our fast generation times implies that we could generate a larger number of solvable levels than the other methods in the same amount of time.

### C. Diversity

Here we compare the diversity of the generated levels using the compression distance metric (the Normal variant as described in Section III), as well as our A* diversity metric. The null hypothesis here is again that the diversity of NoveltyNEAT is comparable to the baselines.

| | Maze | | Mario | |
|---|---|---|---|---|
| | Compression Distance | A* Diversity (Edit) | Compression Distance | A* Diversity (Edit) |
| NoveltyNEAT (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)**$^{*\dagger}$ |
| DirectGA+ | **0.493 (0.002)**$^{\dagger}$ | **0.41 (0.01)**$^{\dagger}$ | 0.55 (0) | 0.33 (0) |
| DirectGA (Novelty) | **0.494 (0.002)**$^{\dagger}$ | **0.40 (0.01)**$^{\dagger}$ | 0.44 (0.01) | 0.20 (0.01) |
| PCGRL (Wide) | **0.525 (0.021)**$^{*\dagger}$ | **0.43 (0.01)**$^{\dagger}$ | 0.56 (0) | **0.55 (0)**$^{*\dagger}$ |
| PCGRL (Turtle) | **0.513 (0.006)**$^{*\dagger}$ | **0.43 (0.01)**$^{\dagger}$ | 0.56 (0) | **0.55 (0)**$^{*\dagger}$ |

Table V: Diversity metrics. Higher is better.

Results are shown in Table V, and we see that NoveltyNEAT'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 compression distance for Mario, we find no statistically significant difference between our method and the baselines.

The A* diversity metric measures DirectGA's Mario levels as quite similar, as the same rough trajectory solves most levels, because they are quite flat, without many jumps or platforms (see Fig. 10c). 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. NoveltyNEAT is somewhere in between, indicating that different trajectories and strategies are required, but the difference, on average, is not as large as PCGRL.

Another point to note is 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.

### D. Difficulty

Next, we investigate the difficulty of our levels, as measured by the leniency metric [20, 21], as well as our A* difficulty metric. We again use the null hypothesis that our method has the same mean difficulty as our baselines.

| | Maze | | Mario | |
|---|---|---|---|---|
| | Leniency | A* Difficulty | Leniency | A* Difficulty |
| NoveltyNEAT (Ours) | 0.70 (0.08) | 0.06 (0.08) | 0.17 (0.23) | 0.24 (0.06) |
| DirectGA | - | - | **1.0 (0)**$^{*\dagger}$ | 0.27 (0) |
| DirectGA+ | **0.60 (0.01)**$^{*\dagger}$ | 0.16 (0.02) | 0.31 (0.01) | **0.14 (0.01)**$^{*\dagger}$ |
| DirectGA (Novelty) | **0.59 (0.01)**$^{*\dagger}$ | 0.16 (0.01) | **0.78 (0.04)**$^{\dagger}$ | 0.24 (0.01) |
| PCGRL (Wide) | 0.64 (0.04) | **0.21 (0.01)**$^{\dagger}$ | 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) |

Table VI: Difficulty. The optimal value usually depends on the player's skill and designer's intentions.

Table VI shows the results. We find that, for the Maze, our method generates quite lenient levels, and this is confirmed by the A* difficulty metric, which has a lower value. For Mario, our levels have quite low leniency, but this varies drastically, so we only find a statistically significant difference between NoveltyNEAT and the DirectGA, which generates very flat levels, and thus has relatively high leniency.

### *E. 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 NoveltyNEAT will be able to generate levels of arbitrary size quickly, without any retraining, as we use a size agnostic method of generation. We test this claim by taking an evolved generator (that was trained only to generate levels of size $14 \times 14$), and simply using it to generate levels of different sizes. We compare this against the DirectGA method, and the results are shown in Figs. 6 and 7.

We find that NoveltyNEAT's generation speed is still substantially higher than the DirectGA's (for both Mario and Maze), even as we increase the level size. 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. This makes sense, as we kept the population size and number of generations constant, and there is work that indicates that normal genetic algorithms do not always perform very well with high dimensional problems [59, 60]. For Mario, we see a slight downward trend in solvability as the level size increases, whereas DirectGA remains constant. This DirectGA consistency in the Mario game 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 make up for this.

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

The above results indicate that NoveltyNEAT has superb generalisation, without requiring retraining, mostly due to how we generate levels. 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, the DirectGA is less appropriate in this situation, as the above factors directly affect generation time, and might increasingly be unsuited for real-time generation, as shown in Fig. 8. Some sample levels from the different methods are shown in Figs. 9 and 10

## VII. Discussion and Future Work

The method introduced here follows a recent trend of focusing more on level generators, which can be applied in real time after an offline training period [10, 11] instead of searching for the levels directly. We demonstrate several advantages over these works, however, 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.

We illustrate some problems with compression distance as a measure of level diversity and introduce a method that does not fall victim to these issues of being dependent on the level size and string representation.

We also introduce a difficulty metric that is based on the state tree of an A* agent, but this showed low correlation with existing metrics.

Future work could include analysing the effects of the different hyperparameters of our generation method, or attempting to make this method more configurable during runtime, by allowing users to specify some desired characteristics, similar to specifying the level size as we do here.

Additional work could investigate the effect of different novelty search distance functions (with a potential focus on simulation-based ones like the A* diversity), or attempt to apply multi-objective optimisation [61] or illumination algorithms [62] to this problem. Another option is exploring the generation of endless levels, where the next parts of the level are generated as the player is playing (possibly using the player's behaviour as input to the network to generate adaptive levels) [10].

Promising avenues for the metrics include comparing the A* diversity metric to the KL-Divergence based score introduced by Shu et al. [10], a neural semantic hashing approach as described by Isaksen et al. [63] or against human judgements of diversity.

Difficulty could also be measured in a more learning-based way, where the difficulty is proportional to how long an agent (e.g. a reinforcement learning agent or evolutionary controller) needs to learn before being able to solve a level, similar to [64]. Different types of agents can even be used to approximate different levels of player skill [65].

## VIII. Conclusion

In conclusion, this work introduces a NEAT and novelty search based level generation approach that (1) does not require any training data, (2) requires minimal game-specific knowledge, can (3) be generally applied and (4) generate levels of arbitrary size rapidly after an offline training period. We compare this method against a standard, 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 mostly in terms of diversity, while generating levels significantly faster than these methods, and requiring less hand-engineered rewards.

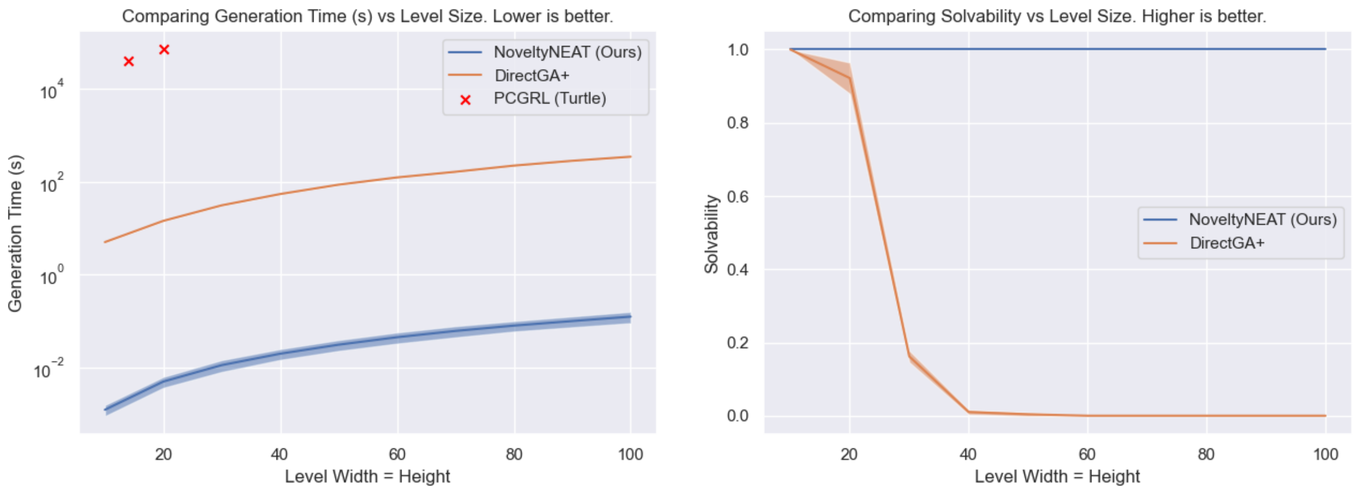
We also introduce two general agent-based metrics that measure the diversity and difficulty of levels. These again do not require any game-specific knowledge or intricate feature representations, but simply a game engine and an agent. Our diversity metric is more expressive than compression distance for the Maze game, and is not as dependent on the size of the levels, or the string representation used.

Source code is publicly released[7].

## Acknowledgements

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

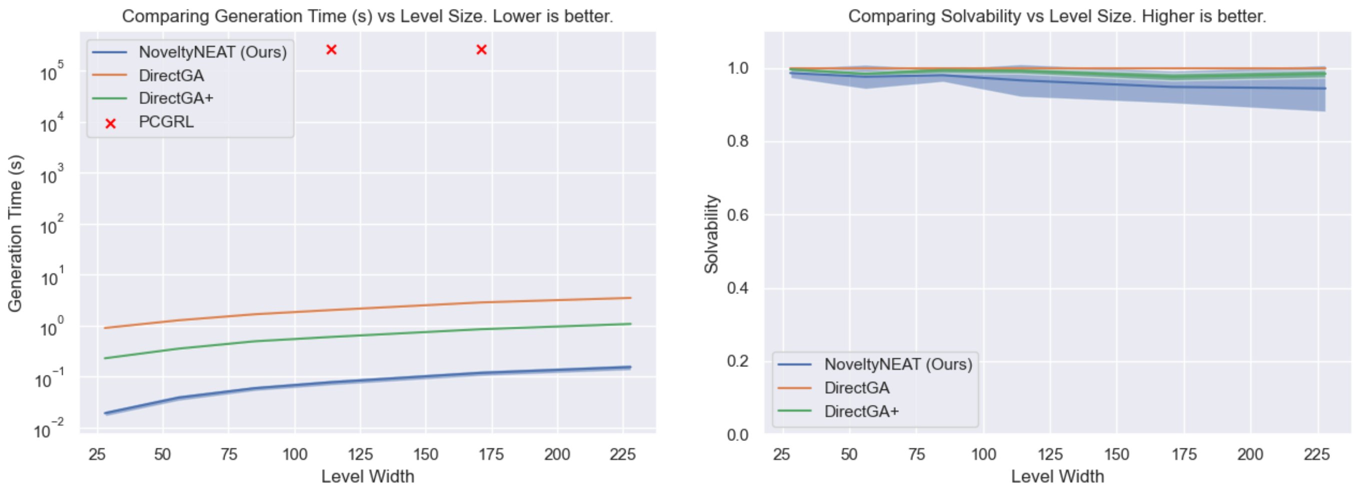
[7] https://github.com/Michael-Beukman/PCGNN



(a) Comparing Maze: NoveltyNEAT vs DirectGA generation time. We only list two PCGRL points, which is how long it took to train $14 \times 14$ and $20 \times 20$ generators respectively. Lower is better

(b) Comparing Maze: NoveltyNEAT vs DirectGA solvability. Higher is better.

Fig. 6: Metrics for Maze levels of different sizes.



(a) Comparing Mario: NEAT vs DirectGA generation time. Lower is better. Since we stopped the PCGRL training after 3 days, we plot this as the amount of time taken.

(b) Comparing Mario: NEAT vs DirectGA solvability. Higher is better.

Fig. 7: Metrics for Mario levels of different sizes.

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

Thank you to Dr. Pravesh Ranchod, who initially introduced the idea of using edit distance to compare two sets of actions.

Thank you to my supervisors, Dr. Steven James and Professor Christopher Cleghorn for providing very useful feedback, acting as a sanity check and contributing ideas for solving some difficult problems.

## References

[1] ESA, “2020 essential facts about the video game industry,” 2020. [Online]. Available: https://www.theesa.com/resource/2020-essential-facts/

[2] D. Pinelle, N. Wong, and T. Stach, “Heuristic evaluation for games: usability principles for video game design,” in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2008, pp. 1453–1462.

[3] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, “Procedural content generation for games: A survey,” *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, no. 1, pp. 1–22, 2013.

[4] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, “Search-based procedural content generation: A taxonomy and survey,” *IEEE Transactions on Compu-*

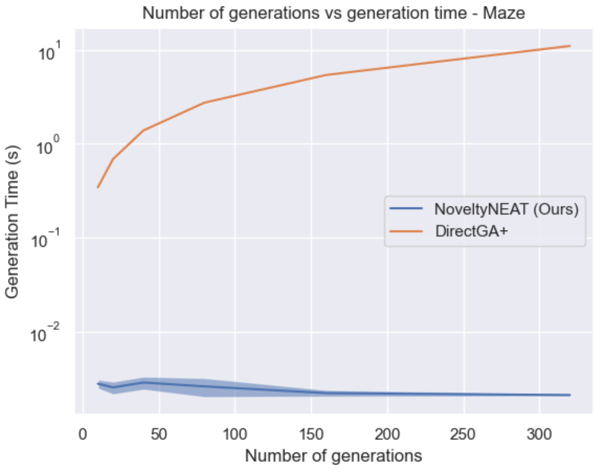


Fig. 8: Maze - Showing the effect of increasing the number of generations on generation time.

*tational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[5] D. Maung and R. Crawfis, "Applying formal picture languages to procedural content generation," in *2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES)*. IEEE, 2015, pp. 58–64.

[6] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural content generation: Goals, challenges and actionable steps," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, vol. 6, pp. 61–75. [Online]. Available: https://doi.org/10.4230/DFU.Vol6.12191.61

[7] N. Sturtevant and M. Ota, "Exhaustive and semi-exhaustive procedural content generation," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, no. 1, 2018.

[8] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (PCGML)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018. [Online]. Available: https://arxiv.org/pdf/1702.00539.pdf

[9] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications*, pp. 1–19, 2020.

[10] T. Shu, J. Liu, and G. N. Yannakakis, "Experience-driven pcg via reinforcement learning: A super mario bros study," in *2021 IEEE Conference on Games (CoG)*. IEEE, 2021, p. accepted.

[11] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "PCGRL: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, 2020, pp. 95–101.

[12] A. Liapis, G. N. Yannakakis, and J. Togelius, "Constrained novelty search: A study on game content generation," *Evolutionary computation*, vol. 23, no. 1, p. 101–129, Mar. 2015. [Online]. Available: https://doi.org/10.1162/EVCO_a_00123

[13] L. Ferreira, L. Pereira, and C. Toledo, "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*, ser. GECCO Comp '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 45–46. [Online]. Available: https://doi.org/10.1145/2598394.2598489

[14] M. Preuss, A. Liapis, and J. Togelius, "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, 2014, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CIG.2014.6932908

[15] J. Schrum, V. Volz, and S. Risi, "CPPN2GAN: Combining compositional pattern producing networks and gans for large-scale pattern generation," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 139–147. [Online]. Available: https://doi.org/10.1145/3377930.3389822

[16] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, "Evolving mario levels in the latent space of a deep convolutional generative adversarial network," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 221–228.

[17] D. Gravina, A. Khalifa, A. Liapis, J. Togelius, and G. N. Yannakakis, "Procedural content generation through quality diversity," in *IEEE Conference on Games, CoG 2019, London, United Kingdom, August 20-23, 2019*. IEEE, 2019, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CIG.2019.8848053

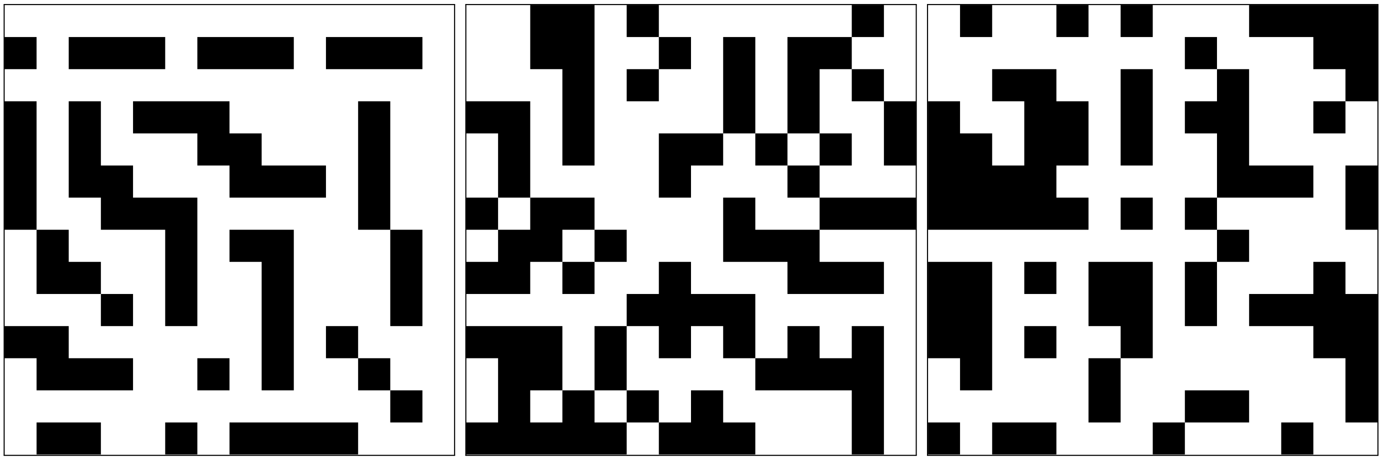
[18] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, p. 99–127, Jun. 2002. [Online]. Available: https://doi.org/10.1162/106365602320169811

[19] J. Lehman and K. Stanley, "Abandoning objectives: Evolution through the search for novelty alone," *Evolutionary computation*, vol. 19, pp. 189–223, 06 2011.

[20] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha, "Launchpad: A rhythm-based level generator for 2-d platformers," *IEEE Transactions on computational intelligence and AI in games*, vol. 3, no. 1, pp. 1–16, 2010.

[21] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'neill, "Evolving levels for super mario bros using grammatical evolution," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2012, pp. 304–311.

[22] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. D. Kraker, "The role of semantics in games and simulations," *Computers in Entertainment (CIE)*, vol. 6, no. 4, pp. 1–35,

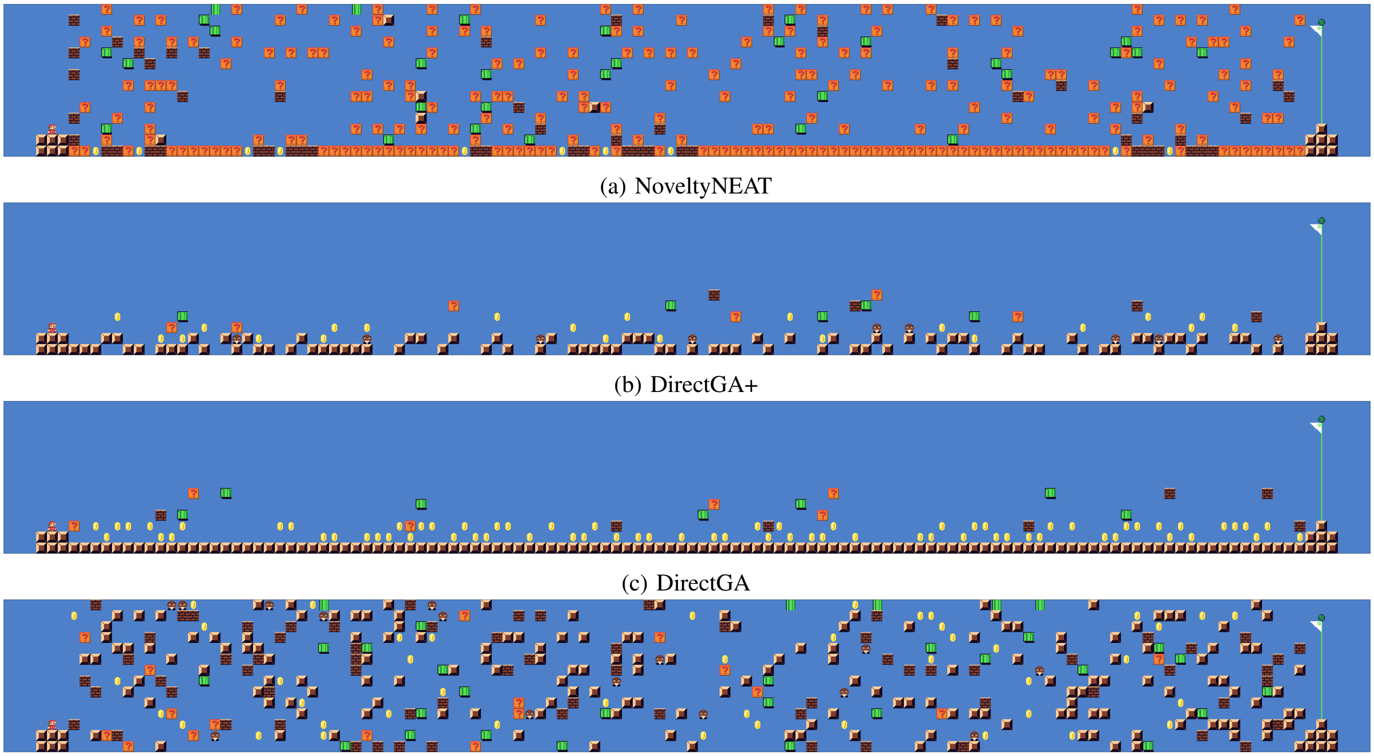


(a) NoveltyNEAT

(b) DirectGA+

(c) PCGRL (Turtle)

Fig. 9: Maze: Example levels.



(a) NoveltyNEAT

(b) DirectGA+

(c) DirectGA

(d) PCGRL (Turtle)

Fig. 10: Mario: Example levels.

2008.

[23] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating generalization in deep reinforcement learning through procedural level generation," *arXiv preprint arXiv:1806.10729*, 2018.

[24] D. E. Goldberg, *Genetic algorithms in search*. Addison Wesley Publishing Co. Inc., 1989, ch. 1.

[25] J. Gomes, P. Urbano, and A. L. Christensen, "Evolution of swarm robotics systems with novelty search," *Swarm Intelligence*, vol. 7, no. 2, pp. 115–144, 2013.

[26] J. Lehman and K. O. Stanley, "Evolving a diversity of virtual creatures through novelty search and local competition," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, 2011, pp. 211–218.

[27] S. Risi, J. Lehman, D. B. D'Ambrosio, R. Hall, and K. O. Stanley, "Petalz: Search-based procedural content generation for the casual gamer," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 244–255, 2016.

[28] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic*

*Programming and Evolvable Machines*, vol. 8, no. 2, p. 131–162, Jun. 2007. [Online]. Available: https://doi.org/10.1007/s10710-007-9028-8

[29] A. Liapis, G. Yannakakis, and J. Togelius, "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*, 07 2013, pp. 343–350.

[30] L. Cardamone, D. Loiacono, and P. L. Lanzi, "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*, ser. GECCO '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 395–402. [Online]. Available: https://doi.org/10.1145/2001576.2001631

[31] M. Kerssemakers, J. Tuxen, J. Togelius, and G. Yannakakis, "A procedural procedural level generator generator," in *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, 09 2012, pp. 335–341.

[32] E. Hastings and K. Stanley, "Evolving content in the Galactic Arms Race video game," in *CIG2009 - 2009 IEEE Symposium on Computational Intelligence and Games*, 09 2009, pp. 241–248.

[33] K. Compton, "Remarks during a panel session at the fdg workshop on pcg," 2010.

[34] N. Shaker, "Intrinsically motivated reinforcement learning: A promising framework for procedural content generation," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[35] D. Vrajitoru, "Large population or many generations for genetic algorithms? Implications in information retrieval," in *Soft Computing in Information Retrieval*. Springer, 2000, pp. 199–222.

[36] N. Shaker, J. Togelius, G. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 Mario AI championship: Level generation track," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 332–347, Dec. 2011.

[37] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, no. 3, pp. 147–161, 2011.

[38] J. Lehman and K. O. Stanley, "Revising the evolutionary computation abstraction: minimal criteria novelty search," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 103–110.

[39] A. Liapis, C. Holmgård, G. N. Yannakakis, and J. Togelius, "Procedural personas as critics for dungeon generation," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2015, pp. 331–343.

[40] Z. Michalewicz and M. Schoenauer, "Evolutionary algorithms for constrained parameter optimization problems." *Evolutionary Computation*, vol. 4, pp. 1–32, 03 1996.

[41] B. Horn, S. Dahlskog, N. Shaker, G. Smith, and J. Togelius, "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, 2014, pp. 1–8.

[42] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitányi, "The similarity metric," *IEEE transactions on Information Theory*, vol. 50, no. 12, pp. 3250–3264, 2004.

[43] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning skills without a reward function," *arXiv preprint arXiv:1802.06070*, 2018. [Online]. Available: https://arxiv.org/pdf/1802.06070.pdf

[44] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[45] N. Shaker, G. N. Yannakakis, and J. Togelius, "Feature analysis for modeling game content quality," in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*. IEEE, 2011, pp. 126–133.

[46] A. K. Hoover, J. Togelius, and G. N. Yannakis, "Composing video game levels with music metaphors through functional scaffolding," in *First computational creativity and games workshop. ACC*, 2015.

[47] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[48] J. Gomes, P. Mariano, and A. L. Christensen, "Devising effective novelty search algorithms: A comprehensive empirical study," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 943–950. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.717.6684&rep=rep1&type=pdf

[49] V. Monga and B. L. Evans, "Perceptual image hashing via feature points: performance evaluation and tradeoffs," *IEEE transactions on Image Processing*, vol. 15, no. 11, pp. 3452–3465, 2006.

[50] A. Hadmi, W. Puech, B. A. E. Said, and A. A. Ouahman, "Perceptual image hashing," in *Watermarking-Volume 2*. IntechOpen, 2012.

[51] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[52] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, Oct. 2011, pp. 289–296.

[53] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[54] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis, "The mario ai championship 2009-2012," *AI Magazine*, vol. 34, no. 3, pp. 89–92, 2013.

[55] S. Shaphiro and M. Wilk, "An analysis of variance test for normality," *Biometrika*, vol. 52, no. 3, pp. 591–611, 1965.

[56] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50 – 60, 1947.

[57] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[58] "Steam hardware & software survey: September 2021," https://store.steampowered.com/hwsurvey/, 2021, [Online; accessed 25-October-2021].

[59] X. Yao and Y. Liu, "Scaling up evolutionary programming algorithms," in *International Conference on Evolutionary Programming*. Springer, 1998, pp. 103–112.

[60] Y. Liu, X. Yao, Q. Zhao, and T. Higuchi, "Scaling up fast evolutionary programming with cooperative coevolution," in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, vol. 2. Ieee, 2001, pp. 1101–1108.

[61] H. Tamaki, H. Kita, and S. Kobayashi, "Multi-objective optimization by genetic algorithms: A review," in *Proceedings of IEEE international conference on evolutionary computation*. IEEE, 1996, pp. 517–522.

[62] J. Mouret and J. Clune, "Illuminating search spaces by mapping elites," *CoRR*, vol. abs/1504.04909, 2015. [Online]. Available: http://arxiv.org/abs/1504.04909

[63] A. Isaksen, C. Holmgård, and J. Togelius, "Semantic hashing for video game levels," *Game Puzzle Des*, vol. 3, no. 1, pp. 10–16, 2017.

[64] D. Ashlock and J. Schonfeld, "Evolution for automatic assessment of the difficulty of sokoban boards," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.

[65] M. González-Duque, R. B. Palm, D. Ha, and S. Risi, "Finding game levels with the right difficulty in a few trials through intelligent trial-and-error," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 503–510.

[66] G. Syswerda, "Simulated crossover in genetic algorithms," in *Foundations of genetic algorithms*. Elsevier, 1993, vol. 2, pp. 239–255.

[67] W. M. Spears *et al.*, "Adapting crossover in evolutionary algorithms." in *Evolutionary programming*, 1995, pp. 367–384.

## Appendix

Here we list the detailed hyperparameters that were used in the above experiments. Implementation details can be found in our source code linked above.

### *NoveltyNEAT*

The hyperparameters for NoveltyNEAT are shown in Table VII.

### *DirectGA*

For the DirectGA, we use 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 use these, as they were unspecified by Ferreira et al. [13], simple to implement, relatively standard [24] and the crossover operations were shown to perform relatively well [66, 67].

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

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

| | DirectGA+ | | DirectGA Novelty | |
|---|---|---|---|---|
| Population Size | 100 | | 50 | |
| Number of Generations | 100 | | 100 | |
| Fitness | $Entropy(DE = 1)$ | $\times 0.5$ | $Entropy(DE = 0)$ | $\times 0.33$ |
| | $PSolvability()$ | $\times 0.5$ | $PSolvability()$ | $\times 0.33$ |

Table VIII: Maze hyperparameters for DirectGA

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

| | 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 |
| Coins Block Size | 10 | 10 | 10 |
| Blocks Block Size | 40 | 10 | 10 |
| Ground Maximum Height | 2 | 2 | 2 |

Table IX: Mario hyperparameters for DirectGA. Explanations of these parameters is provided by Ferreira et al. [13].

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 was weighted equally to the normal fitness function.

### *PCGRL*

For PCGRL, as previously mentioned, we trained for 100 million timesteps for Maze, 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. [11], 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[8], 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.

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

| | 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 | |
| $\lambda$ | 0 | | 0 | |
| Novelty Distance Function | Visual Diversity, only on reachable tiles | | Visual Diversity | |
| Fitness | $Novelty()$ | $\times 0.399$ | $Novelty()$ | $\times 0.25$ |
| | $Solvability()$ | $\times 0.202$ | $Solvability()$ | $\times 0.50$ |
| | *Intra-Novelty*$(neighbours = 10)$ | $\times 0.399$ | *Intra-Novelty*$(neighbours = 2)$ | $\times 0.25$ |

Table VII: Hyperparameters for NoveltyNEAT