Hierarchically Composing Level Generators for the Creation of Complex Structures

Michael Beukman, Manuel Fokam, Marcel Kruger, Guy Axelrod, Muhammad Nasir, Branden Ingram, Benjamin Rosman, Steven James

***Abstract*—Procedural content generation (PCG) is a growing field, with numerous applications in the video game industry and great potential to help create better games at a fraction of the cost of manual creation. However, much of the work in PCG is focused on generating relatively straightforward levels in simple games, as it is challenging to design an optimisable objective function for complex settings. This limits the applicability of PCG to more complex and modern titles, hindering its adoption in industry. Our work aims to address this limitation by introducing a compositional level generation method that recursively composes simple low-level generators to construct large and complex creations. This approach allows for easily-optimisable objectives and the ability to design a complex structure in an interpretable way by referencing lower-level components. We empirically demonstrate that our method outperforms a non-compositional baseline by more accurately satisfying a designer's functional requirements in several tasks. Finally, we provide a qualitative showcase (in Minecraft) illustrating the large and complex, but still coherent, structures that were generated using simple base generators.**

I. Introduction

Procedural content generation (PCG) is a research field focused on automatically generating game content, such as levels, maps and music [1]. PCG has several benefits: it is often cheaper than manually designing content [2], and it allows for significantly more content to be generated than would otherwise be possible through human creation [3, 4]. PCG has recently garnered more attention [5], leading to impressive results in numerous games [6, 7, 8].

However, much of the research done in the field of PCG focuses on simple, 2D tilemap games, such as *Super Mario Bros*, *The Legend of Zelda*, and maze games [8, 9, 10, 11, 12]. While these are useful testbeds, they are often very simple, and do not exhibit complex structures. This focus on simplicity makes it non-trivial to generalise these methods to more complex and modern games. Furthermore, few methods focus on, or excel at, generating large and complex levels, which is a necessary step for PCG to become more mainstream [13].

This has led to recent research that applies PCG to more complex titles [14, 15, 16, 17], particularly Minecraft. Minecraft is a 3D, discrete voxel-based game, making it amenable to grid-based PCG methods originally developed for 2D tilemaps. It is also more complex than these aforementioned games and thus provides a useful domain to bridge the gap between traditional and modern games.

Despite progress in the field, there are still several shortcomings to current methods. Some rely on a developer manually designing level components [18], which can be costly [2] and difficult to transfer to new games [10]. Other approaches generate levels that lack core functional requirements, such as being playable [14]. Lastly, some methods generate relatively small and simple structures [16, 17], limiting their applicability to complex games.

Our work aims to address these limitations by proposing an approach that decomposes the problem of generating a complex and large-scale level into smaller, but manageable pieces. For instance, it is simple to generate an abstract, high-level layout of a town (by considering houses, roads and gardens as the individual building blocks) while ensuring all houses are reachable via roads. Similarly, it is relatively straightforward to learn to generate a single house, garden, or farm. Composed together, however, these simple generators can easily generate a fully-fledged, and functional, town.

In our method, multiple generators are hierarchically composed to generate large and complex structures. This is inspired by *hierarchical reinforcement learning* [19], where a complex task is broken down into smaller components and an agent learns how to optimally perform each subcomponent. We break the construction of a high-level structure (e.g., a town) into simple subtasks (e.g., houses, gardens, etc.) and train independent agents to generate these individual components. Furthermore, instead of manually designing level elements, our method allows a designer to specify an objective function, i.e., what constitutes a "good" component.

This approach has three main benefits: (1) due to the independence, users can modify the creation of each component (at different levels of abstraction) in isolation; (2) the combinatorial explosion in the number of levels that can be generated [20]; and (3) the ability to specify the desired structure by referencing subcomponents, allowing easier generation of complex levels compared to normal, non-hierarchical methods.

We extend PCGNN [9], a recent method which uses neuroevolution and novelty search to evolve diverse level generators, by hierarchically composing these independent generators to create large and complex structures in tile-based games. We demonstrate that using composition makes it significantly easier to generate complex structures compared to using a single, non-compositional generator, an effect which becomes more pronounced as we increase the complexity of the low-level structures. Furthermore, we can automatically generate large-scale towns and cities by specifying a few, simple objectives.[1]

School of Computer Science and Applied Mathematics, University of the Witwatersrand.

[1]Code is available at https://github.com/Michael-Beukman/MCHAMR

II. Background

A. Hierarchical and Compositional Reinforcement Learning

Reinforcement learning (RL) [21] is a field focused on solving sequential decision-making problems, where an agent interacts with an environment (e.g., by placing specific tiles in a tilemap level) and receives a scalar reward signal indicating how good an action was (e.g., obtaining a high reward when removing a tile that creates a path in a maze [10]).

However, *long-horizon tasks* require a long sequence of actions to solve (e.g., generating a level by sequentially placing many tiles) and are often challenging for standard RL methods [19, 22]. Hierarchical RL addresses this by decomposing complex tasks into (1) smaller subtasks and a (2) top-level policy that selects which subtask to perform [19, 23]. In compositional RL, agents compose low-level skills to perform more complex behaviours, allowing tasks to be specified in a more understandable way by referencing these skills [20].

B. Evolutionary Algorithms

Many PCG methods [1, 8, 11] use aspects of evolutionary computing, a subclass of optimisation algorithms that attempt to mimic natural selection [24]. Genetic algorithms (GAs) usually consist of a collection (or population) of individuals, each possessing a *genome* which describes the individual (e.g., an integer vector representing the height of a platform [11]). These genomes are mapped to *phenotypes*, which can be thought of as the result or instantiation of this individual in the actual problem domain (e.g., a level generated using the given height information). To create the next generation, the current population's high-performing individuals (determined by the *fitness function*) are merged through *crossover* to produce children, which are then subjected to slight mutations.

1) NeuroEvolution of Augmenting Topologies (NEAT): NEAT [25] is a genetic algorithm that evolves the weights and structure of neural networks. NEAT starts with a population of simple networks with no hidden layers, which gradually increase in complexity through evolution, to better solve the problem under consideration.

2) Novelty Search: Novelty search [26] rewards individuals based on their novelty compared to the population, instead of their objective performance (as in traditional genetic algorithms). This results in improved exploration and can help to solve hard or deceptive[2] problems.

C. PCGNN

PCGNN [9] is a method that uses NEAT [25] and novelty search [26] to evolve level generators. The main goal of this work is to learn reusable level generators that can quickly generate multiple different levels, as opposed to searching for single levels each time one is desired. Each individual in this population is a neural network, which can be used to generate multiple levels. To obtain a level from a network, an initial, random, tilemap is created. Then, for each tile, the neural network receives as input the surrounding tiles and outputs the current center tile. This process is repeated, sequentially, for each tile. PCGNN generally uses multiple fitness functions: (1) the normal novelty search objective, to incentivise exploration; (2) intra-generator novelty, which incentivises one neural network to generate multiple diverse levels; and (3) one or more *feasibility* fitness functions, which describe the feasibility criteria for the specific game (e.g., solvability in a maze). PCGNN generates levels quickly and can generalise to unseen level sizes, allowing the generation of arbitrarily-sized levels without retraining.

[2]Deceptive problems are those where directly maximising fitness can lead to being stuck in local minima [26, 27].

III. Related Work

A. General PCG

Although there are numerous approaches to PCG, many focus on generating simple levels, and are therefore difficult to generalise to more complex settings, such as modern games. Search-based techniques, such as evolutionary algorithms, are commonly used to generate levels that maximise a specific objective function [1, 8, 9, 11, 17]. Similarly, the recent paradigm of Procedural Content Generation via Reinforcement Learning generates levels using RL, training an agent policy to sequentially place tiles to maximise some reward [10, 16, 28]. However, it is challenging to design a monolithic objective or reward function that incentivises the generation of complex and functional structures while being optimisable [13]. Other methods rely on a dataset of existing content (which may not be available for many games), applying machine learning to generate novel content [29, 30, 31].

1) Generating Complex Levels: While many approaches focus on simple 2D tile-based games, such as *Super Mario Bros* and *The Legend of Zelda*, other work rather aims to generate more complex content. One notable example is the *Generative Design in Minecraft (GDMC)* competition, where the task is to generate an entire settlement in Minecraft [15, 18]. This competition aims to improve the generation of holistic and complex constructions (e.g., an entire, coherent town instead of just a single building) while taking the external environment into account. Many submissions to this competition, however, focused on more hand-designed, rule-based [17] methods specifically designed for settlement generation, which may require significant effort to generalise to other scenarios. This has led to work that focuses on automatically generating lower-level structures one would find in a settlement, such as buildings. For instance, Green et al. [32] use constrained growth algorithms to separately generate the building and floor plan, which can be useful when customisation is desired. Barthet et al. [17] use DeLeNoX [33], to evolve diverse building generators for Minecraft using an autoencoder-based novelty score. Here, the focus is on generating creative buildings, rather than generating complex and functional structures or how to combine these structures into an actual settlement.

2) Adversarial and Open-Ended Approaches: Recently, PCG has also been used to generate a large number of diverse levels on which machine learning agents can be trained, leading to these agents becoming more robust [34, 35]. Level generators and game-playing agents can often work in tandem, where the generator generates levels that challenge the agent.

As the agent improves, the level generator must also improve to generate more complex and harder levels [7, 36].

Most of these methods, however, focus on simple games (such as mazes [37] or 2D terrains [38]), making it unclear how to generalise them to more complex games. Furthermore, the focus is often on obtaining robust and high-performing artificial agents [37] instead of generating video game levels to be played and enjoyed by humans. Prioritising the former could lead to overly difficult levels. These levels may also have different characteristics compared to human-designed ones [39], complicating their use as a substitute for manual content creation. Lastly, treating the generation process as one large problem may limit the ability of the generator to create levels with enough complexity to challenge the machine learning agent. Instead, having multiple generators that solve simpler problems at different levels of abstraction could enable the generation of more complex and larger-scale structures [40].

B. Compositional PCG

Much work has also been done on combining different PCG methods to generate more complex pieces of content, or even entire games [41]. For instance, Togelius et al. [42] combine an Answer Set Programming (ASP)-based approach [43] with a genetic algorithm that searches over variables for the ASP. Liapis et al. [41] advocate for game design orchestration, where different content generators are combined to generate full, coherent, games. One benefit of this compositional approach is allowing algorithms to specialise in (and thus excel at generating) specific pieces of content. There are still numerous open problems in this field, however, such as how to best coordinate automated generators and human designers.

Hierarchy can also be used to improve the generation speed of PCG methods. For example, Smith and Bryson [44] find that using ASPs to generate large levels is infeasibly slow but that using hierarchy can lead to much faster generation. They first generate the high-level structure of a level and subsequently fill in the details within this fixed structure. However, this approach uses hand-designed high-level structures, which may not be easily generalisable to new games [13].

Snodgrass and Ontanon [40] also leverage hierarchy to generate structures at different scales. From a set of example levels, they extract a set of high-level and low-level patterns and fit two multi-dimensional Markov chains to this data. To generate a level, then, the high-level model generates the structure while the low-level generator fills in the details. This method results in improved generation compared to a non-hierarchical approach, but requires example levels as training data, which may not always be available [29].

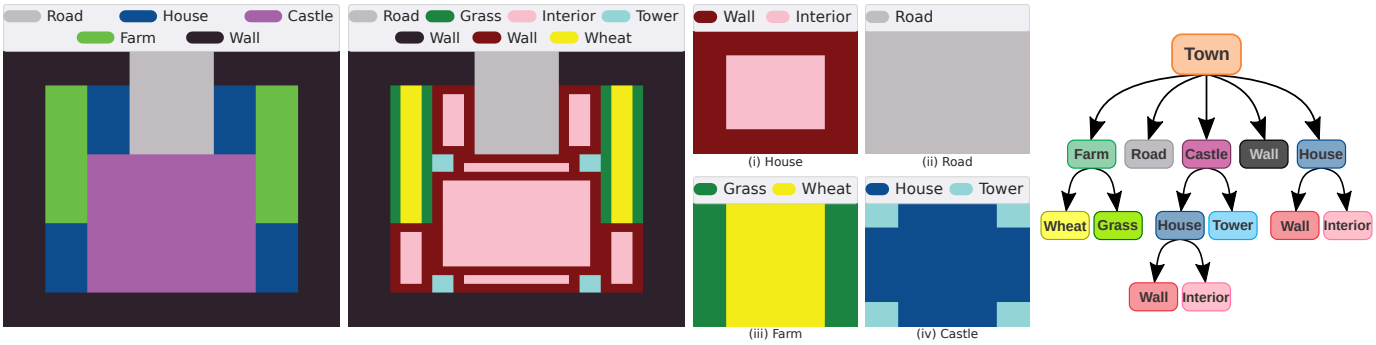
Hierarchy can also be used to separately construct different aspects of a level. For instance, Dormans [45] first generates the "mission": the high-level task that the player must complete. The physical layout of the level is then generated based on the mission. Similarly, in *Unexplored* [46], a graph representing the level is created by applying a set of replacement rules to create, remove and modify nodes to obtain certain properties. This graph is then mapped onto a grid, and each node is transformed into a room [47]. Although this approach is hierarchical, each of the 5000 replacement rules is hand-designed, hindering application to other settings.

Another use of composition is to increase the resolution of high-level sketches provided by a designer. For instance, Liapis et al. [48] allow a designer to draw a high-level, low-resolution sketch of a level. Then, using techniques such as genetic algorithms, the PCG system replaces each high-level tile with a more detailed and fleshed-out component. While this can be used to obtain a high-resolution version of a designer's sketch, it may be less suitable when a complex structure consisting of several specific components is desired. Instead of hand-designing the high-level sketch, Liapis [49] allows the designer to specify a fitness function that is used to evolve it. The sketch is then parsed into a graph indicating which segments are connected, thereby defining constraints for each segment (such as the number and location of doors). Using these constraints, each segment is evolved and placed in the high-level sketch to form a complete level. While this work is promising, it focuses on dungeon levels, without a clear way to apply the same technique when generating other types of levels. Furthermore, this approach directly evolves the sketch and the low-level segments, meaning that evolution occurs at generation time, which may be prohibitively slow [9].

Dormans and Leijnen [50] take a different perspective: their two-step process first generates a large amount of diverse content, and the second step reorganises this content into an actual level. While this simplifies the generation step, the content is reorganised without any hierarchical elements.

Other methods create levels by generating and combining different *layers*, effectively composing together different generators. However, these methods generally focus on iteratively furnishing a level, as opposed to hierarchically generating a complex layout. For instance, Green et al. [51] first generate the structure of a dungeon level—tiles such as walls and the floor. Then, a different method furnishes this structure by placing items such as treasure and enemies. Also following a similar layered approach, Wu et al. [52] generate natural levels. They break the level down into several layers such as ocean, land, and forests, with each layer being generated by a cellular automaton (CA). This approach is a promising way to generate landscapes, but the handcrafted rules could be difficult to generalise to other domains. Furthermore, although CAs can generate landscapes, they may be less well-suited to generating structures that satisfy certain complex constraints—for instance, cities where houses must be reachable by roads. *Dwarf Fortress* [53] also uses multiple layers; here, the world, history and several other aspects evolve over time in a rule-based simulation. This process begins by first generating the world and progressively adding more specialised designed systems. The world is created by generating an elevation map using a randomised fractal, followed by multiple layers of maps including temperature, rainfall and vegetation. Once the world is created, the simulation of civilisation begins where settlements are built, trade routes are formed, and wars are fought. The simulation stops at a designated point in time, and the player starts the game in a unique living world.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2023.3297619



(a) High-level view of a town. (b) The town after composition. (c) Base components. (d) The tree structure for (b).

Figure 1: An example of MCHAMR, with the (a) high-level and (b) composed towns and (c) the base components. The tree structure defining how components are combined is shown in (d), with leaf nodes corresponding to low-level tiles.

IV. Hierarchically Composing Level Generators

Here we describe our approach, **M**ulti-level **C**omposition of **H**ierarchical and **A**daptive **M**odels via **R**ecursion (MCHAMR).

To illustrate our method, we first consider the example of generating a simple medieval town. At a high level, we can represent the town as a 2D tilemap, shown in Figure 1a, with houses, a large castle and roads. Each of these components, however, can be decomposed into simpler components; for instance, a house that is a single tile in Figure 1a actually consists of external walls and an empty interior, as shown in Figure 1c. Using our composition algorithm (described below), we can use the definitions of these smaller structures together with the high-level map to obtain a fully-fledged and detailed town, shown in Figure 1b. Furthermore, the user has control over each component. Should they wish to make a similar town but would prefer to replace or modify a specific structure, this is possible without altering any of the other components.

To implement this method in an automatic way, we require the concept of a *generator* $g \in G$. This object can generate a tilemap W, which is represented as a 2- or 3-dimensional matrix, where each element is taken from some tile set T. Each generator optionally has a *tile mapping* $M_g : T \to G$ that controls which subgenerators are used when this generator places a specific tile. For instance, in the above example, the town generator's mapping specifies that the "house" tile actually maps to the generator shown in the top left pane of Figure 1c. In essence, we construct a tree (Figure 1d shows the tree for the previous example), where each node is a level generator, with parent nodes generating placeholders to be filled by their children and leaf nodes producing the lowest-level, non-abstract, structures. To generate the level, we start at the root and traverse through the tree.

Finally, for each generator g, we also have S_g, which indicates the size of the subtiles of this generator, i.e., the size at which the next level of the hierarchy will be generated. For example, if our town generator g_{town} produces an abstract tilemap of size 10×10, and has an associated $S_{g_{town}}$ of 3×3, then the final level will have size 30×30. Each tile in the abstract map corresponds to a 3×3 block of low-level tiles.

Given a set of generators, our algorithm (detailed in Algorithm 1) works as follows: We start with the top-level generator, which generates a tilemap (line 4 in Algorithm 1). In standard PCG, the procedure would terminate here. We instead go further and interpret each tile as an instruction to use another generator and place its output at that location (lines 14-16). Each of these lower-level generators can also be composed of generators. If at any point, though, the generator does not contain a *tile mapping* M_g, we end the recursion (lines 5-6). The overall generation process is illustrated in Figure 2.

This method tends to produce blocky structures since each lower-level tile is of the same size. To overcome this, we combine connected tiles of the same type into a single, larger, "logical" tile (line 12). Then, at generation time, we generate one large structure of this size, instead of many small ones. To achieve this, we find and merge contiguous rectangles by attempting to expand the dimensions of each tile one at a time until we can no longer do so. We repeat this process at every level of the hierarchy, coalescing connected tiles before recursively calling the subgenerators. Coalescing not only improves the cohesiveness of the generated structure but also gives the top-level generator more control over the scale of the lower-level components. Figure 3 illustrates this process, and Appendix D contains some example levels.

Now, if we are interested in using MCHAMR with automatic level generators, they must satisfy some requirements. Specifically, the generators must generate levels quickly (as we need to generate many low-level structures); be able to generate structures of arbitrary sizes (as the sizes of the final level and its components are not fixed beforehand); and generate many diverse levels (to prevent the different structures from looking identical, and thus uninteresting, to a player).

Although our approach is agnostic to the exact level generators used, one method that is particularly well-suited to act as the low-level generator is PCGNN, which satisfies all of our requirements. Additional benefits include that the training time is generally low [9] and, while we can include detailed domain knowledge, this is not required.

A. Using PCGNN in MCHAMR

Having presented the workings of MCHAMR, we next describe a concrete approach that uses PCGNN to train the low-level generators. First we train each component separately,

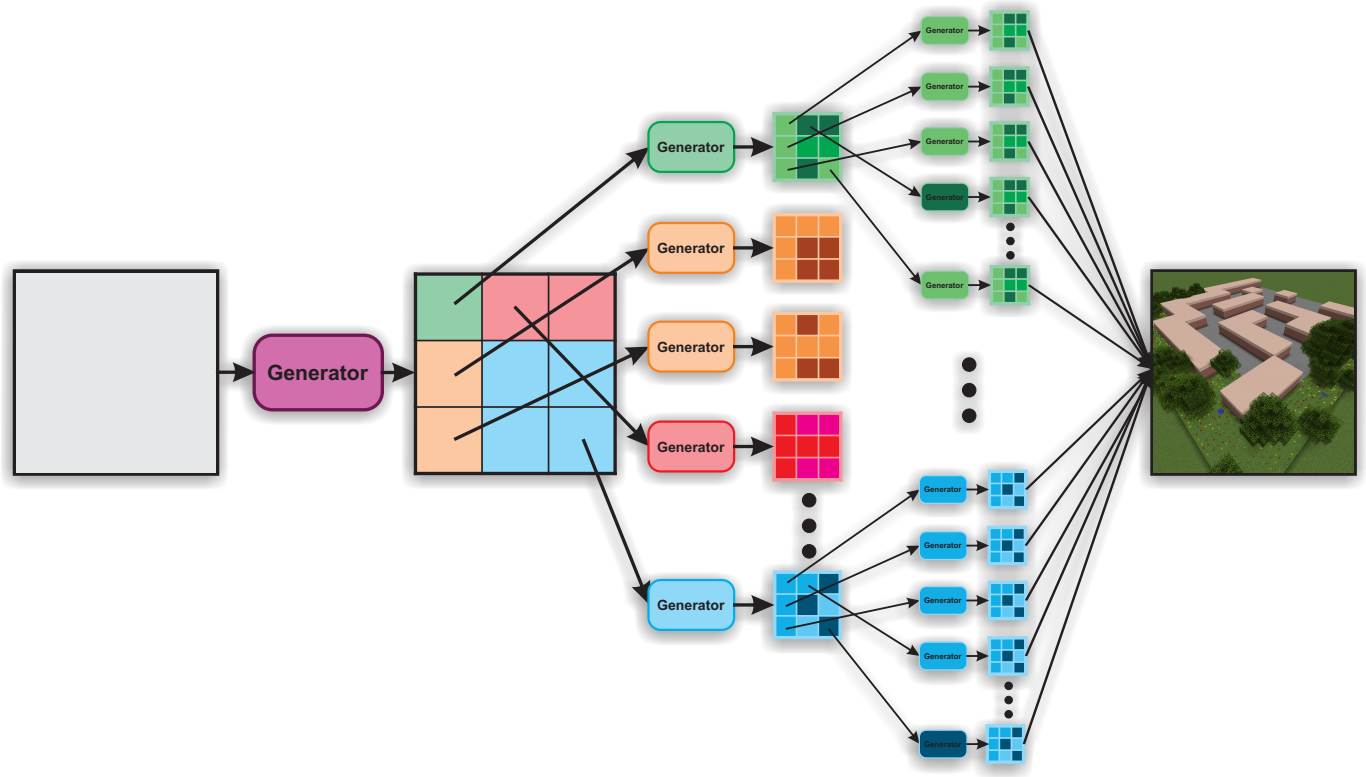
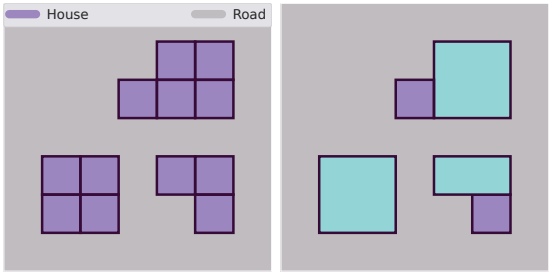


Figure 2: A high-level illustration of MCHAMR. We begin with a top-level generator, which generates an abstract tilemap. Each tile in this map is an instruction to use a lower-level generator and to place its output at that location. Each of these generators can be further composed of other generators. In the end, we obtain a fully-fledged level.



(a) (b)

Figure 3: (a) A level, and (b) the result after coalescing connected tiles. Cyan tiles were coalesced into one larger tile.

using some objective function that describes a feasible individual. Once each of these generators has been trained, we can compose them together, needing only to specify which tiles map to which generators. This has multiple benefits; for instance, we can train and design multiple generators in parallel, obtaining a repertoire of components that can be composed later on. The approach is also modular, so we could leverage the same generator for different purposes (e.g., a house could be used in many different scenarios).

Additionally, we can also train generators faster in this factored way compared to training one monolithic generator. Consider building an $n \times n$ level that consists of some high-level structure (e.g., an abstract map consisting of single-tile houses and roads) and low-level components (e.g., a generated house). Let us compose the $n \times n$ level out of a high-level structure of size $\sqrt{n} \times \sqrt{n}$ and low-level components of size $\sqrt{n} \times \sqrt{n}$. Now, during training, the flat method must generate levels of size $n \times n$. For the composed method, however, we

Algorithm 1 MCHAMR

```
// We call this initially with the top-level generator, as well as the desired size of the final level.
procedure GENERATE(g ∈ G, size ∈ ℤ² ∪ ℤ³)
    // Generate a map, possibly abstract, using g
    map = MAKEMAP(g, size/S_g)
    if M_g is null then
        return map // Return map if this is the lowest level
    end if
    // Otherwise recurse to the next level in the hierarchy
    // Create an empty map of sufficient size
    FilledOutMap = Empty(size)
    // For each position, tile and coalesced size in the map
    for all pos, tile, S_c ∈ COALESCE(map) do
        // Recursively generate using the appropriate size
        NewMap = GENERATE(M_g(tile), size = S_g × S_c)
        // Place this generated map at the correct location
        FilledOutMap[pos×S_g : (pos+S_c)×S_g] = NewMap
    end for
    return FilledOutMap
end procedure
```

have two separate training procedures, each generating levels of size $\sqrt{n}\times\sqrt{n}$. Thus, the overall time complexity[3] of training the flat method would be $O((n \times n)) = O(n^2)$, whereas the composed method would be $O(2(\sqrt{n} \times \sqrt{n})) = O(n)$. For large levels, this speedup in training time can be quite significant. Note that during inference, however, each method generates the same amount of tiles.

Furthermore, as there is no dependence between different level generators during training, we can redefine the meaning of a specific generator. For instance, a town generates a tilemap containing houses, roads and gardens. We could also change the interpretation of the individual tiles, resulting in the same generator generating cities that consist of towns, highways and parks. This effectively constructs multiple different tree structures, with the same generator as the root and different child generators. In general, this approach would work as long as the same feasibility criteria are valid in both cases.

Finally, while PCGNN is particularly suited to be used as the low-level generator in `MCHAMR`, this is by no means a requirement. We could use other suitable level generators (e.g., [8, 28, 31]), hand-designed components, or a combination of these. This provides a significant amount of control and flexibility, which is useful in designing a game [54].

V. Experiments and Results

A. Experimental Setup

1) Domains: We consider two domains to demonstrate the capabilities of `MCHAMR`. We first quantitatively demonstrate the effects of composition in a simple 2D town-building game. We next experiment on Minecraft to demonstrate that our approach can also be applied to more complex 3D games.

2) Experiments: We perform three main experiments in this section. In Section V-B, we first illustrate the benefits of our hierarchical generation approach as the complexity of the lower-level structures increases. The second experiment, in Section V-C, directly compares our hierarchical method against a non-compositional, flat approach. Finally, in Section V-D, we qualitatively evaluate our method by generating complex towns and cities in Minecraft. More details about our PCGNN implementation and our experiments, including hyperparameters and exact fitness functions, are listed in Appendices A and C, respectively. In our quantitative experiments, we assess the extent to which each method fulfils the fitness functions used during optimisation. This enables us to measure how leveraging hierarchy can streamline the optimisation process, resulting in improved fitness. Achieving a higher fitness value corresponds to the generator more faithfully realising the designer's intended outcome.

B. Hierarchical Generation

A compositional approach allows us to simplify the task of the top-level generator by abstracting away the details of the lower-level components. In particular, for a compositional approach to build a house, it requires a single action—placing the high-level house tile. A non-compositional approach, however, requires a large number of coordinated actions—placing each of the low-level tiles that make up the house. In this experiment we compare these two paradigms, using a simple town level with houses, gardens and roads.

To this end, we introduce the concept of *window size*, a proxy for how complex the lower-level structures are. For instance, a window size of 1×1 means that placing a single tile is sufficient to generate a high-level structure, corresponding to the compositional setting. A larger window size, such as 2×2, means that a house requires 4 coordinated actions to build, which corresponds to a non-compositional approach. Similarly, 5×5 means that a house requires 25 coordinated actions, i.e., the low-level structures are harder to build.

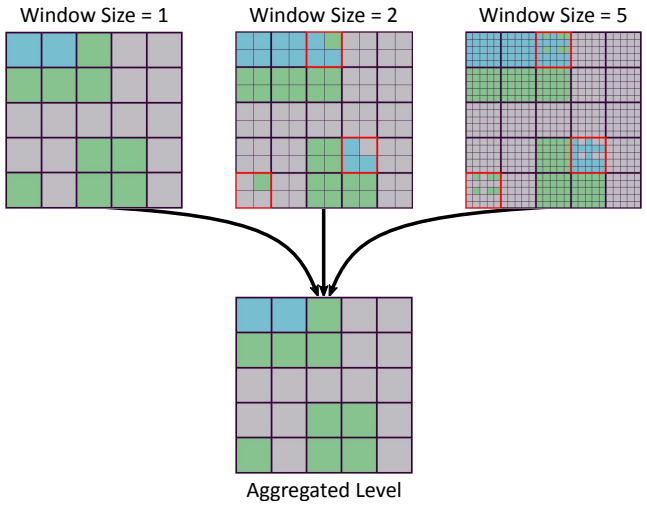


Figure 4: The top row illustrates three generated levels, with window sizes of $1\times1, 2\times2$ and 5×5, respectively. The bottom row shows the aggregated level, i.e., the level that each of the above ones reduces to. The red squares indicate logical tiles that were collapsed to the default tile of grass due to not being fully populated with the same tile. Here, blue tiles are houses, green tiles are gardens and grey corresponds to roads.

We implement this as follows (see Figure 4 for an example): For a window size of 2×2, say, we consider each non-overlapping 2×2 window as a single tile. If all the tiles within a window of a certain size are identical, then a single tile of that type is placed in the aggregated level. Otherwise, the tile is replaced with a "default" tile—gardens in this case. While we could change this rule (e.g., taking the majority tile), our choice simulates that a certain number of coordinated actions *must* be performed to generate a single component. By contrast, in the compositional case, only one action is required. In essence, the larger the window size, the more coordinated actions are required to generate a single tile. We also consider window sizes of 3×3, 4×4, 5×5 and 10×10, to evaluate how the complexity of the low-level structures influences our final performance. Note that we generate levels with an aggregated size of 10×10 tiles. This means that, for a window size of 2×2, the number of total tiles is 20×20, which is then aggregated into a 10×10 level (where each 2×2 block in the large level is compressed to a single tile).

[3]We omit factors such as the population size and the number of generations as they are kept constant for both approaches.

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2023.3297619

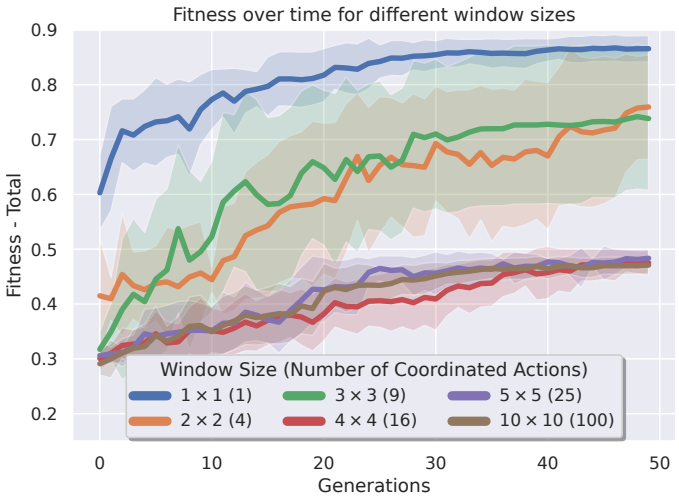


Figure 5: A plot showing the maximum fitness value in the population (i.e., how "correct" the generated levels are) over time for different window sizes (i.e., how many coordinated actions are required to build the lower-level structures). The mean over 10 seeds is shown, with standard deviation shaded.

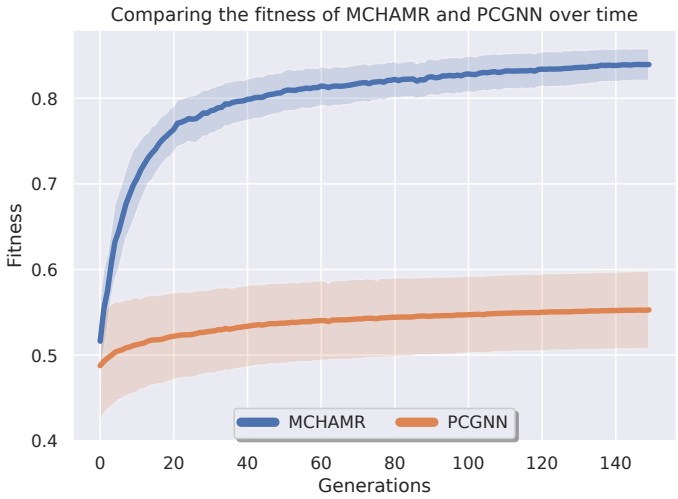


Figure 6: Illustrating the fitness over time on 20 random town layouts for MCHAMR (consisting of the composed town and house generators) compared to the flat, non-compositional method, corresponding to vanilla PCGNN. We first compute the average fitness across 10 seeds and plot the mean and standard deviation over the 20 layouts.

This experimental setup has the added benefit of being comparable, as each method uses the same fitness functions, and the only difference is in the size of the generated structures. The fitness functions used here are (1) reachability, where houses must be reachable via roads, and (2) a probability fitness, incentivising a roughly equal number of house, garden and road tiles. See Appendix C for more details.

We compare the fitness of each method over time in Figure 5, which indicates that using the compositional approach results in a higher final fitness value compared to the other, non-compositional methods. The performance decreases significantly when we have a much larger 10×10 window size (corresponding to an overall level size of 100×100), where each house requires $10^2 = 100$ coordinated actions to build. Even this is still significantly fewer actions than a real Minecraft house. Overall, the results show that simplifying the task of the top-level generator by abstracting away the details of the lower-level components is beneficial.

C. Composition vs Flat Generation

Our next experiment directly compares MCHAMR against a non-compositional baseline that generates the entire level using PCGNN. In particular, we are again interested in building towns, this time attempting to replicate a specific, randomly generated, town layout.

For the compositional approach, we learn two separate generators for houses and towns, while the non-compositional method generates the entire level at once. In particular, we have a 25×25 level, consisting of a collection of 5×5 houses. The high-level town structure also has a size of 5×5. The fitness functions used did not include novelty, and measured the average overlap between the desired level and the generated one. The desired levels use one set layout for houses, where walls surround empty space, and a randomly generated town layout. We generate 20 of these random town layouts, train a separate generator on each of them, and average the fitness values obtained.

The fitness results over time are shown in Figure 6, where it is clear that PCGNN's fitness cannot touch that of MCHAMR. Our composed method achieves significantly higher fitness, and generates much more accurate levels than the flat method. See Appendix C for examples of the generated levels.

D. Minecraft Showcase

Finally, we evaluate our method on Minecraft and show a few qualitative examples of interesting and complex structures that were generated by MCHAMR. We use the Evocraft [14] library to place and visualise the generated structures in game.[4]

Here we generate 3D settlements, where each town consists of houses, gardens and roads. Additionally, to demonstrate the notion of reusing a single generator, we further generate *cities* using the town generator, by redefining houses to be towns. Concretely, we use the following generators, with Appendix C discussing the fitness functions in more depth.

Town A high level 2D tilemap of a town. We used several objectives, but (similarly to Section V-B) primarily rewarded reachability and having roughly an equal number of house, road and garden tiles.

House A 3D house, incentivised to have a roof, walls and an empty interior. The house also had novelty and intra-generator novelty objectives.

Garden A 2D tilemap with flowers, grass, ponds and trees. The generator is incentivised to generate levels with at least one of every tile, no more than 5% pond tiles, between 20% and 70% grass tiles and trees that are not too close to each other. Gardens and houses share the same novelty objectives.

[4]https://github.com/real-itu/Evocraft-py

This article has been accepted for publication in IEEE Transactions on Games. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TG.2023.3297619

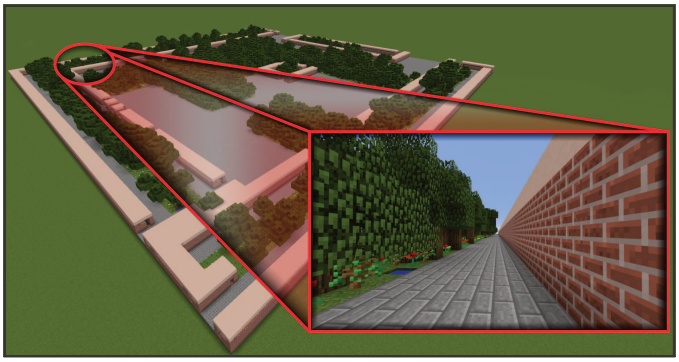


Figure 7: A generated city and a zoomed-in portion of an alleyway. The city and town generators are the same.

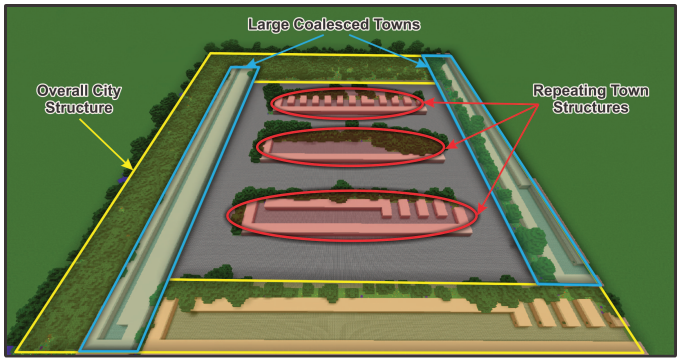
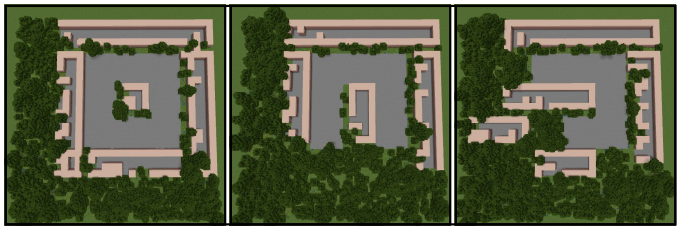


Figure 8: An annotated example of a city.

(a) Three levels from generator A.



(b) Three levels from generator B.

Figure 9: Showing three levels each from two generators (a) and (b). Each generator comes from the same experiment, just with different initial random seeds. Generator A has large swaths of gardens, with some towns around the center. Generator B generally has gardens in the bottom and left sides, with towns in the middle and along the top and right edges.

In Figure 7, we use the same generator for the city and the town. We also show a first-person view of a specific section to illustrate the detail in each part of the generated settlement.

Figure 8 shows an annotated example of a generated city. The large, high-level structure of the city is generated by the top-level generator. Then, each tile that it places is transformed into a town by the town generator. We see that coalescing results in longer towns and the towns themselves, while not identical, contain similar repeating structures.

In Figure 9, we illustrate the benefits of incentivising novelty during training. In this figure, we use the same houses and garden generators as before, alongside a town generator that was trained using four fitness functions: (1) Intra-generator Novelty; (2) Novelty Search; (3) Reachability; and (4) a Probability fitness incentivising roughly an equal number of houses, gardens and roads. We use the tile-pattern KL-Divergence metric, with a pattern size of 2×2, as the novelty distance function [55] and weigh the fitness functions with ratios $1 : 1 : 4 : 4$. We generate cities using the same neural network as the towns. These levels show that if we use the same generator to generate multiple levels, we can obtain qualitatively different structures while still adhering to a certain style. Furthermore, running the same training procedure multiple times—with the same hyperparameters but a different initial random seed—can result in very different structures. Details about training, novelty objectives and a comparison of novelty scores can be found in Appendix E.

Finally, we note that the structures we generate are very large in the scale of standard PCG. For instance, *Super Mario Bros.* levels have size 114×14. Much of the work in Minecraft also generates relatively small structures, such as $7 \times 7 \times 7$ mazes [16] or $20 \times 20 \times 20$ lattices [17]. The composed towns we generate can easily be $200 \times 5 \times 200$ or larger, totalling more than $100\ 000$ blocks. Despite this massive scale, we are still able to generate coherent structures and sensible towns. Large settlements are also generated for the GDMC competition [15], but unlike our approach, these generations often require significant hand-designing effort [18].

VI. Discussion

The approach introduced here, MCHAMR, follows a recent trend that emphasises the importance of more complex and open-ended creations in PCG [14, 15, 17]. We break the generation problem down into separate components, and train a model for each component. This is different from much of the contemporary research on generative models, where methods tend to train one monolithic model. We believe that there are several good reasons for this factorisation. First, by factorising each salient component of a level, each of these can be changed separately without altering anything else. For instance, suppose we have a good town generator that we are satisfied with, but wish to change the house layout. If we had one large model, it would need to be entirely retrained with an alternative objective function, with no guarantee that the other pieces would indeed remain unchanged. On the other hand, when we factorise the components, each component can be changed independently. Secondly, if we have separate generators, each generator's fitness function is much simpler

to create (for the designer) and also simpler to optimise (for the learning algorithm). By contrast, it can be challenging to design a single monolithic objective function for complex tasks such as building a town. Even if we can create such an objective, it would likely be hard to optimise by the agent.

Our approach does require a designer to specify the components and desired hierarchy, as well as particular fitness functions. We believe that this gives the designer immense control over the generation, as they can completely specify which components should be used, and how they should be combined. Additionally, we believe that designing a fitness function for each component—while requiring some effort—is an elegant way to allow designers to specify what is defined as good, without needing to design the components themselves.

Our method has numerous applications, such as (partially) generating large open-world games. This would allow designers freedom in specifying certain, reusable low-level components that can be composed in numerous different ways, thereby enticing players towards exploration.

Furthermore, in contrast to many other methods, we abstract away details relevant to a particular game by having separate high- and low-level generators. Separating these responsibilities could enable the abstract generators to be used in a variety of games, needing only to swap out the low-level generators for each game. Relatedly, since we independently train the generators, it would be possible to have a community-driven database of useful generators, which can then be seamlessly composed together to generate novel artifacts. This could be similar to *Picbreeder* [56],[5] where users collectively contribute to the generation of many interesting images.

VII. Limitations and Future Work

Our method is just a first step in this direction, and there are numerous avenues for future work. One promising avenue would be to use a combination of different PCG methods as low-level generators [8, 10, 16], instead of just PCGNN. For instance, we could use data-based approaches to generate the low-level components (using, e.g., existing house datasets), while generating the high-level structure using a fitness-based method. Furthermore, while our method is designed particularly with tile-based games in mind, we believe that it could be extended to other types of games by leveraging more appropriate low-level generators. Our tile-based focus may additionally lead to discretisation artifacts, as components can only be placed on a rectangular grid. While this simplifies the hierarchical computation, it may lead to blocky-looking structures. Coalescing is one way to partially alleviate this problem: by generating at a higher resolution and coalescing similar tiles, we could obtain smoother shapes. Future work could also explore other techniques to address this limitation.

While the coalescing rules we used were intentionally simple, they can be made arbitrarily complex; for instance, a designer may wish to only coalesce under certain conditions, and leave most connected tiles as single components.

In future work, we would like to explore the use of quality-diversity algorithms [57] to obtain a collection of diverse and high-performing low-level generators, and use these interchangeably to obtain more diverse content. Furthermore, since we use a 2D generator as the root, the generated levels do not have any verticality. Adding vertical aspects to the levels (e.g., mountains, multi-story houses, etc.) would thus be valuable.

Finally, we have a clear separation between each component without any explicit communication between components, be it at the same or across different hierarchical levels. Thus, all structure must come from the top-level generator, similar to the top-down approach outlined by Liapis et al. [41], and no emergent structure is generated via the interaction of the low-level generators. This may carry some disadvantages, such as neighbouring components being oblivious of one another, leading to incoherent generations.

However, in our approach, this problem could be circumvented by adding more low-level components, and expanding the tileset of the top-level generator—thereby increasing its complexity. The tradeoff here is that this may require more human effort in designing these additional generators (via their fitness functions), and altering the existing hierarchy.

Despite these limitations, this independence enables us to learn modular components that can be used in several settings, whereas reuse would be more difficult if there is a tight coupling between components. Furthermore, we allow the designer to specify the structure and the high-level generator in charge of implementing it. This gives the designer more control compared to a case where the only structure is obtained by the unpredictable interaction of the low-level generators.

Relatedly, independently training the generators does not enable them to learn how to best combine their generated components, or for one generator to make up for the shortfall of another. Instead, the designer specifies the hierarchy and fitness functions for each component. This shifts some of this responsibility onto the designer, but avoids the problem where the high- and low-level generators are incompatible. Future work could consider modifications to this, where generators are jointly trained. This idea could also relate to more open-ended avenues, where the generated structures increase in complexity over time, as opposed to reaching an endpoint when the feasibility fitness is maximised [14, 17]. One way to achieve this could be learning how to dynamically compose different generators, or using learned objective functions.

VIII. Conclusion

We introduce `MCHAMR`, a compositional approach to level generation, which leverages multiple simple generators to generate large and complex structures. Our approach has multiple benefits, such as (1) being configurable, allowing the designer to choose which low-level generators are used; (2) it being straightforward to combine different generators, without needing to specify one monolithic fitness function; and (3) simplifying each generator's task by decomposing the problem, potentially allowing for less training or smaller (and thus faster) models. We demonstrate that our approach improves generation compared to a non-compositional approach, and that it is able to generate large-scale and complex settlements in Minecraft. Ultimately, we hope that this work is a step

[5]https://nbenko1.github.io/

towards more complex and large-scale generations, which is necessary for the adoption of PCG in the gaming industry.

ACKNOWLEDGMENTS

Computations were performed using HPC infrastructure provided by the MSS unit at the University of the Witwatersrand. We thank the reviewers for their insightful comments, which helped to strengthen the final version of this paper.

REFERENCES

[1] J. Togelius, G. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Trans. Comput. Intell. AI Games.*, no. 3, 2011.
[2] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, 2013.
[3] G. Smith, "Procedural content generation: An overview," *Level Design Processes and Experiences.*, 2017.
[4] O. Korn, M. Blatz, A. Rees, J. Schaal, V. Schwind, and D. Görlich, "Procedural content generation for game props? A study on the effects on user experience," *Comput. Entertain.*, vol. 15, no. 2, 2017.
[5] N. Brewer, "Computerized dungeons and randomly generated worlds: From rogue to minecraft [scanning our past]," *Proceedings of the IEEE*, vol. 105, no. 5, pp. 970–977, 2017.
[6] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, "Evolving mario levels in the latent space of a deep convolutional generative adversarial network," in *GECCO*. ACM, 2018.
[7] L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, "Adversarial reinforcement learning for procedural content generation," in *CoG*. IEEE, 2021.
[8] S. Earle, J. Snider, M. C. Fontaine, S. Nikolaidis, and J. Togelius, "Illuminating diverse neural cellular automata for level generation," in *GECCO*. ACM, 2022.
[9] M. Beukman, C. W. Cleghorn, and S. James, "Procedural content generation using neuroevolution and novelty search for diverse video game levels," in *GECCO*. ACM, 2022.
[10] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "PCGRL: procedural content generation via reinforcement learning," in *AIIDE*. AAAI, 2020.
[11] L. Ferreira, L. T. Pereira, and C. F. M. Toledo, "A multi-population genetic algorithm for procedural generation of levels for platform games," in *GECCO*. ACM, 2014.
[12] 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 Trans. Comput. Intell. AI Games.*, no. 4, 2011.
[13] J. Togelius, A. Champandard, P. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural content generation: Goals, challenges and actionable steps," in *Artificial and Comput. Intell. in Games*, 2013.
[14] D. Grbic, R. B. Palm, E. Najarro, C. Glanois, and S. Risi, "Evocraft: A new challenge for open-endedness," in *Applications of Evolutionary Computation*. Springer, 2021.
[15] C. Salge, M. C. Green, R. Canaan, and J. Togelius, "Generative design in minecraft (GDMC): settlement generation competition," in *FDG*, 2018.
[16] Z. Jiang, S. Earle, M. C. Green, and J. Togelius, "Learning controllable 3D level generators," *arXiv preprint arXiv:2206.13623*, 2022.
[17] M. Barthet, A. Liapis, and G. Yannakakis, "Open-ended evolution for Minecraft building generation," *IEEE Trans. Games.*, 2022.
[18] C. Salge, M. C. Green, R. Canaan, F. Skwarski, R. Fritsch, A. Brightmoore, S. Ye, C. Cao, and J. Togelius, "The AI settlement generation challenge in Minecraft: First year report," *Künstliche Intelligenz*, 2020.
[19] S. Pateria, B. Subagdja, A. Tan, and C. Quek, "Hierarchical reinforcement learning: A comprehensive survey," *ACM Comput. Surv.*, 2021.
[20] G. N. Tasse, S. D. James, and B. Rosman, "A boolean task algebra for reinforcement learning," in *NeurIPS*, 2020.
[21] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction*. MIT Press, 1998.
[22] O. Nachum, H. Tang, X. Lu, S. Gu, H. Lee, and S. Levine, "Why does hierarchy (sometimes) work so well in reinforcement learning?" *arXiv preprint arXiv:1909.10618*, 2019.
[23] B. Hengst, *Hierarchical Approaches*. Springer, 2012.
[24] D. E. Goldberg, *Genetic Algorithms in Search*. Addison-Wesley, 1989.
[25] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, 2002.
[26] J. Lehman and K. Stanley, "Abandoning objectives: Evolution through the search for novelty alone," *Evolutionary Computation.*, 06 2011.
[27] L. D. Whitley, "Fundamental principles of deception in genetic search," in *Foundations of genetic algorithms*. Elsevier, 1991, vol. 1.
[28] S. Earle, M. Edwards, A. Khalifa, P. Bontrager, and J. Togelius, "Learning controllable content generators," in *CoG*. IEEE, 2021.
[29] 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 Trans. Games*, 2018.
[30] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Comput. Appl.*, vol. 33, no. 1, 2021.
[31] T. Shu, J. Liu, and G. Yannakakis, "Experience-driven PCG via reinforcement learning: A super mario bros study," in *CoG*. IEEE, 2021.
[32] M. C. Green, C. Salge, and J. Togelius, "Organic building generation in minecraft," in *FDG*. ACM, 2019.
[33] A. Liapis, H. P. Martínez, J. Togelius, and G. Yannakakis, "Transforming exploratory creativity with DeLeNoX,," in *ICCC*. ACC, 2013.
[34] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating generalization in deep reinforcement learning through procedural level generation," in *NeurIPS Workshop on Deep RL*, 2018.
[35] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," in *ICML*. PMLR, 2019.
[36] P. Bontrager and J. Togelius, "Learning to generate levels from nothing," in *CoG*. IEEE, 2021.
[37] J. Parker-Holder, M. Jiang, M. Dennis, M. Samvelyan, J. Foerster, E. Grefenstette, and T. Rocktäschel, "Evolving curricula with regret-based environment design," in *ICML*. PMLR, 2022.
[38] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, "Paired open-ended trailblazer (POET): endlessly generating increasingly complex and diverse learning environments and their solutions," *arXiv preprint arXiv:1901.01753*, 2019.
[39] A. Dharna, J. Togelius, and L. B. Soros, "Co-generation of game levels and game-playing agents," in *AIIDE*. AAAI, 2020.
[40] S. Snodgrass and S. Ontanon, "A hierarchical mdmc approach to 2d video game map generation," in *AIIDE*, vol. 11, no. 1. AAAI, 2015.
[41] A. Liapis, G. Yannakakis, M. J. Nelson, M. Preuss, and R. Bidarra, "Orchestrating game generation," *IEEE Trans. Games.*, no. 1, 2019.
[42] J. Togelius, T. Justinussen, and A. Hartzen, "Compositional procedural content generation," in *Workshop on PCG in Games, FDG*, 2012.
[43] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Trans. Comput. Intell. AI Games.*, 2011.
[44] A. J. Smith and J. J. Bryson, "A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games," in *50th Anniversary Convention of the AISB*, 2014.
[45] J. Dormans, "Adventures in level design: generating missions and spaces for action adventure games," in *Workshop on PCG in games*, 2010.
[46] Ludomotion, "Unexplored," [PC Game], Delft, The Netherlands, 2017.
[47] BorisTheBrave, "Dungeon generation in unexplored," https://www.boristhebrave.com/2021/04/10/dungeon-generation-in-unexplored/, April 2021, [Online; accessed 14-April-2023].
[48] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient world: Human-based procedural cartography," in *EvoMUSART*. Springer, 2013.
[49] A. Liapis, "Multi-segment evolution of dungeon game levels," in *GECCO*. ACM, 2017, pp. 203–210.
[50] J. Dormans and S. Leijnen, "Combinatorial and exploratory creativity in procedural content generation," in *Workshop on PCG for Games*, 2013.
[51] M. C. Green, A. Khalifa, A. Alsoughayer, D. Surana, A. Liapis, and J. Togelius, "Two-step constructive approaches for dungeon generation," in *FDG*. ACM, 2019.
[52] Z. Wu, Y. Mao, and Q. Li, "Procedural game map generation using multi-leveled cellular automata by machine learning," in *ISAIMS*, 2021.
[53] T. Adams and Z. Adams, "Dwarf fortress," [PC Game], Bay 12 Games, 2021. [Online]. Available: http://www.bay12games.com/dwarves/
[54] G. Lai, W. H. Latham, and F. F. Leymarie, "Towards friendly mixed initiative procedural content generation: Three pillars of industry," in *FDG*. ACM, 2020.
[55] S. M. Lucas and V. Volz, "Tile pattern kl-divergence for analysing and evolving game levels," in *GECCO*. ACM, 2019.
[56] J. Secretan, N. Beato, D. B. D'Ambrosio, A. Rodriguez, A. Campbell, J. T. Folsom-Kovarik, and K. O. Stanley, "Picbreeder: A case study in collaborative evolutionary exploration of design space," *Evolutionary Computation.*, 2011.
[57] J. Mouret and J. Clune, "Illuminating search spaces by mapping elites," *arXiv preprint arXiv:1504.04909*, 2015.