Generation of diverse Minecraft structures with evolutionary algorithms

Ben Hutchings

School of Computing Science, Newcastle University, UK

Abstract

Write your abstract here. This should be a concise summary of what your project has been about and what you aim to show. 9pt text here.

Keywords: list the main concepts used in your project, e.g. the research area and techniques used.

Table of Contents

Introduction

Procedural content generation (PCG) is a technique used in video game design and other creative fields, such as art and music. The technique allows creators to generate a near infinite amount of content based on a few rules, constraints, and parameters. This allows games to be highly replayable, feeling more unique and unpredictable. An example of effective PCG is in No Man's Sky (Games 2016), a universe exploration game, which can generate 2^{64} ($18 * 10^{18}$) worlds, each ~203 km^2 (Newhouse 2016). Compared to one the biggest non-generated games, ARMA 3 (Interactive 2012), which has a map size of 270 km^2 (Carlson 2013), shows how much more content can be generated by PCG.

PCG does come with some downsides, mainly sacrificing quality control. A traditional game would have game designers' hand-designing all aspects of the game, meaning the output is predetermined and easily controllable. Since PCG is designed to be unending, it is impossible to brute force quality check. Unlike other types of PCG, like art and music, the design of content in games is first and foremost based around playability, with artistic qualities being a secondary objective. Many existing machine learning PCG techniques allow for small variations in the content generated, which in a medium like music makes no difference to the overall quality, but small variations in game design can lead to the game being unplayable. But playability is not a binary scale, as (Koster 2013) explains, there exists a "golden section" of game design which is the perfect balance novelty and familiarity, which is pleasing for our brains and therefore better to play.

Another drawback to PCG is processing costs and generation speed. According to the steam hardware survey (Steam 2023), which is the most comprehensive hardware census and continuously updated, there is still a massive range in hardware capabilities being used, with strongest graphics card being ~2400% stronger than the weakest (UserBenchmark 2023). The higher the hardware requirements for a game, the larger the proportion of users unable to run the game, alienating a portion of the customer base.

An approach to PCG is using machine learning. Outside of games, machine learning based PCG has made massive leaps forward. Models such as DALL-E 2 (Ramesh et al. 2022) can generate completely unique photo-realistic images from a text description, and ChatGPT (OpenAI 2022) can generate text answers to almost any question. These models have been so successful due in large part to the enormous amount of data they have been trained on and the huge complexity of the models, which presents problems for PCG in games. For DALL-E/ChatGPT to generate whatever is requested, they needed to have seen something similar to base their response off of. This is problematic for game world generation, as there exists no dataset big or diverse enough to adequately train these models (Summerville et al. 2018). Large-scale deep learning models are also extremely computationally expensive, heavily affecting the hardware requirements.

Another issue is the idea of creativity. Models such as DALL-E and ChatGPT do produce unique outputs, but the output is based on many existing sources. For these models this is not a big problem because of the huge variety of training data which allows the models to draw on many sources to generate new content ("The Quest for Ai Creativity" 2015). Games on the other hand generally have the same recognizable styles and artifacts throughout the world. A model splicing these together could lead to a disjointed world and recognizable from other games, leading to lower immersion.

The fundamental problem with PCG is the the *state-explosion* problem, where as the size of the generated content increases, the search space increases exponentially and brute force solutions become intractable (Godefroid and Khurshid 2002). Since it is impossible to verify if a solution is the best solution, an approximate solution is required. One approach to this is using genetic algorithms (GA). GA's take inspiration from the biological process of natural selection and use it to evolve a solution to a problem, using heuristics to guide a population towards a solution. Evolving a solution instead of training from past solutions comes with certain advantages. By having an algorithm which is controlled by the rules of a system rather than being trained on existing examples allows the model to come up with new unique solutions, rather than rehashing existing solutions.

This paper will be experimenting with NEAT (Stanley and Miikkulainen 2002), which is a type of genetic algorithm, and how the performance is affected by different levels of novelty. I will be investigating this within the context of the EvoCraft (Grbic et al. 2020) challenge, a PCG challenge for MineCraft (Persson 2023). Theses concepts will be explained in more detail.

Aims and Objectives

Aim: Investigate how varying levels of novelty affects the diversity of procedurally generated structures in Minecraft Objectives:

1. Investigate other competition entries for the EvoCraft challenge.
2. Research procedural content and novelty search techniques.
3. Implement fitness functions to evaluate single structures.
4. Experiment with varying levels of novelty.
5. Evaluate multi-structure cities to see the effects of novelty.

Background

Minecraft

Minecraft is a 3D, open-world, sandbox, voxel-based video game. Each voxel, called a block, can be broken and replaced to build structures, allowing players to apply their creativity. An extract of blocks and a structure are shown in Figure 18. Minecraft uses PCG and world seeds to create a unique world which is 3.6 billion blocks^2 (Whitworth 2021), allowing players virtually infinite space to explore and build. Because of the open-ended nature of the game and the simplicity of interactions with the world, Minecraft has become a platform for many AI challenges, including mineRL. This competition focused on an agent within Minecraft which has to complete a variety of tasks in an unknown environment. Because there is no one defined task, the algorithm has to be able to complete many smaller problems, with the eventual goal of improving research into general intelligence.

For interactions between Python and Minecraft world I will be using a Minecraft server with a Bukkit (2010) plugin called RaspberryJuice installed and the McPi Python library to communicate with it. Bukkit is a server modification tool with an API which allows developers to easily create server plugins. The RaspberryJuice plugin converts Python commands into Java commands which can be processed by the Minecraft server. The McPi library simply sends Python commands to the RaspberryJuice plugin, which then places them in the Minecraft environment.

The EvoCraft Challenge

To investigate genetic algorithms for PCG, I am using on the EvoCraft Challenge as a framework to base the project on. The EvoCraft challenge brief is to create an open-ended algorithm which is capable of creating novel and increasingly complex structures in MineCraft. These algorithms have to be unending and should aim to diverge over time rather than slow down and become repetitive. One of the drawbacks of PGC was the lack of quality control, and the problem with infinitely generating content becoming repetitive over time. This challenge aims to use evolving algorithms to keep generating content which keeps diverging and becoming more interesting.

Other EvoCraft Entries

Evocraft PCGNN

(Beukman et al. 2023) were came runners up in the EvoCraft competition with their endless city generator. Their approach broke a city down into the component 'levels', starting from the lowest level, the house and garden. To generate a house and garden they broke this down into 4 components: the house structure, roof, decorations, and garden. They then used a PCGNN (Procedural Content Generation using Neat and Novelty search) approach to generate each of these components. The house, roof, and decorations are all generated as 3d tilemaps to be placed in-world. The house consists of walls, empty space, and entrance, the roof consists of a design covering the area beneath it, and the decorations consists of decoration blocks filling floor space inside the house. The garden works slightly differently as it is a 2d tilemap covering an area with flowers, grass, and trees. They then used these component houses and gardens to create a town. A town is its own generated 2D tilemap of houses, gardens and roads, where roads connect all the houses in the town. They then placed many towns together to create a city, which could grow endlessly. Example of this process shown in Figure 19.

simple_minecraft_evolver

(real-itu 2021) The simple Minecraft evolver is basic NEAT implementation which aims to build a tower towards a gold block in the air. Early generations start by building towers in random directions, but as they evolve, they move more towards the gold block.

Neural Networks

A neural network is a type of machine learning technique which is inspired by the human brain. A neural network consists of collections of neurons, organized in layers, which are connected to each other. An example is shown in Figure 1, where each neuron is depicted as a node on the graph and the connections between neurons as vertices. In a traditional neural network, each neuron (apart from the input neurons) has a weighted connection to some nodes from the previous layer (n_{prev}). The equation for the value of a neuron is shown in Equation 1, where W_i is the weight to a node, X_i is the value of the node, and b is a bias.

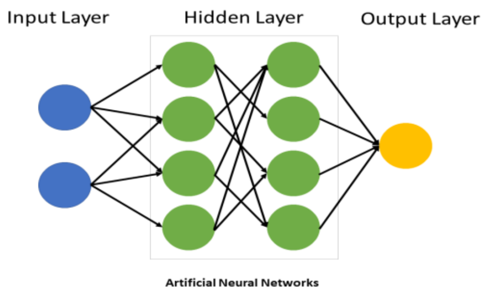


Figure 1 - Example Neural Network

$$y = f\left(\sum_{i=0}^{n_{prev}} w_i\, x_i + b\right)$$

Equation 1- Neuron Equation

To make a prediction, some input values are set to the input nodes values, which are used to calculate the value of the next layer's neurons. This repeats until the output layer is reached. The values from the output layer is the prediction. A traditional neural network can learn by adjusting the weights and bias in the neurons.

NEAT + Novely Search

NeuroEvolution of Augmenting Topologies (NEAT) is type of genetic algorithm (GA) which heavily mimics biological evolution to increase the complexity of neural networks. Unlike traditional neural networks, where the structure of the network is fixed, NEAT works by increasing the complexity of the network over time. NEAT uses a direct encoding method where all nodes, weights, and biases from each individual are all encoded into a "genome", which can be used to recreate the neural network (the phenotype). To begin many individuals created, called a population. These initial individuals have no hidden nodes and are just weighted connections between the input and output nodes. Each individual is evaluated using a fitness function and given a fitness score. The top individuals with the highest fitness score are taken and used to create a new population. To create a new population the individuals undergo "crossover". This is where two genomes are combined to create an "offspring" by randomly combining their genomes together. This process is not done blindly though, as this is much more likely to create a non-functioning neural network rather than improving the performance. To crossover two genomes they first have to share a similar enough genome to allow them to crossover. This is done by recording each network's evolution history and if the genomes share a similar enough history then they can crossover. Once offspring have been created through crossover, they undergo mutation. Mutation is the process of making random changes to an individual's genome, which can show as new nodes or changes to weights in the phenotype. The aim of NEAT is, through random mutation and crossover, to evolve (make random changes) a solution to a problem (Heidenreich 2019).

There is however a problem with this evolution process, a population's fitness score can easily become stuck in a local maxima. This comes from a deeper ideological difference between biological evolution and genetic algorithms. The purpose of an GA is to reach a global maximum fitness, whereas biological evolution aims to both evolve fit individuals which will survive, but will also spread out and diverse over generations. If a species does not diversify then it is more vulnerable to diseases and sudden changes to the environment (Greenwald 2009). For an GA to get out of a local maxima it must first drop in fitness score before finding another genome structure which would allow it to reach the global maxima. Traditional GA algorithms don't allows for this as any drop in fitness score would kill the individual and stop it from evolving further. Here we take a note from biological evolution and promote diversity, called novelty search. Novelty search gives a higher fitness to individuals which have significantly changed from the population. This lets individuals change and explore the search space without being killed, leading to a higher probability of success. (Stanley and Miikkulainen 2002)

Reinforcement Learning PCG

Reinforcement learning (RL) is a type of machine learning composed of three key elements: an agent, the environment, a reward. RL uses a trial-and-error method with an agent interacting with an environment. An agent makes an action within and environment and will either be rewarded, if the the action was a positive action, and punished, if the action was negative. The eventual goal for a RL agent is to learn a policy, which is a mapping from input states to output actions. The agent starts with random actions and getting experience, in the form of state-action-reward, which is used to update the policy of the agent. Eventually the agent aims to maximize the cumulative reward signal by maintaining a balance between exploring, to learn new experiences, and exploitation, leveraging existing techniques.

RL gains many of the same benefits as genetic algorithms. Just like GA's, RL's require no large dataset and therefore don't have any of the intrinsic biases and creativity issues which come with it. There are some subtle differences between the two algorithms. RL can sometimes suffer trying to reach a global maxima in reward/fitness. Because RL is one agent learning over many iterations, if the agent chooses a strategy which works well, but only reaches a local maxima, it would have to unlearn that entire strategy and come up with a new one for it to reach a global maxima. A GA on the other hand has many individuals in a population, each which can explore their own routes (promoted through novelty search). Each route that ends in a local maxima will be killed in favour of a route which produces a global maxima, therefore having a higher likelihood of reaching the global maxima (Joy 2019). While the RL model approach has the potential to be effective in this project, the properties of NEAT make it more desirable.

General Adveserial Networks

A General Adversarial Network (GAN) is type of deep learning network, composed of two neural networks: A generator and discriminator. The role of a generator is, once trained on training data, to generate more examples which resemble the training data. The role of the discriminator is to tell the difference between examples from the training data and the examples generated by the generator (Park, Lee, and Yu 2022).

GAN's have some advantages which make them very powerful in certain situations. Since the training data is only used for examples, there is no need to label the data. Once a GAN is fully trained, both parts of the network can be used. A well trained generator can generate data very similar to the training data. DALLE-2, a realistic image generator, uses a GAN-like network structure to create unique photo-realistic images. The discriminator can also be used to detecting discrepancies in data, for example medical imaging, quality control, and fraud detection. There are some issues with GAN's. They require vast amounts of training data and the wider the range of outputs being generated, the higher the size of the training data required. They are also a black box and very hard to reason why it came up with what it did, making it very hard to fix problems like diversity of output and garbled data. GAN's naturally lose detail from the input to the output, which makes the generated content lose some finer details (Hui 2018). This is not a problem for creative works as some minor variation has no affect on the quality of the output, but does have an effect on PCG for games. Some minor changes to game generation can leave it unplayable and useless. The combination of not being able to generate at a very fine level, being a black box, and requiring huge amounts of training data, it can be very hard to reliable produce quality content for games. Because of these disadvantages a GAN approach is unsuitable for this project.

Implementation

I have chosen to do this project in Python. While the competitors (Java, C++, R…) are much faster than the core Python modules, there are libraries, such as NumPy, which are written in C and are extremely quick. This means it can perform well on large datasets, making it a favorite for data scientists. Because it is a popular data science language, there is also a large collection of machine learning libraries supported to help development.

The aim of the project is to procedurally create diverse Minecraft structures. Taking a note from (Beukman et al. 2023), the structure generation is broken down into 2 parts, generating the base of the house and generating the roof, an example of this is in Figure 2. For each of these parts, a different model is used. The development of these models is broken down into three main objectives: model design, the fitness functions, and novelty experimentation.

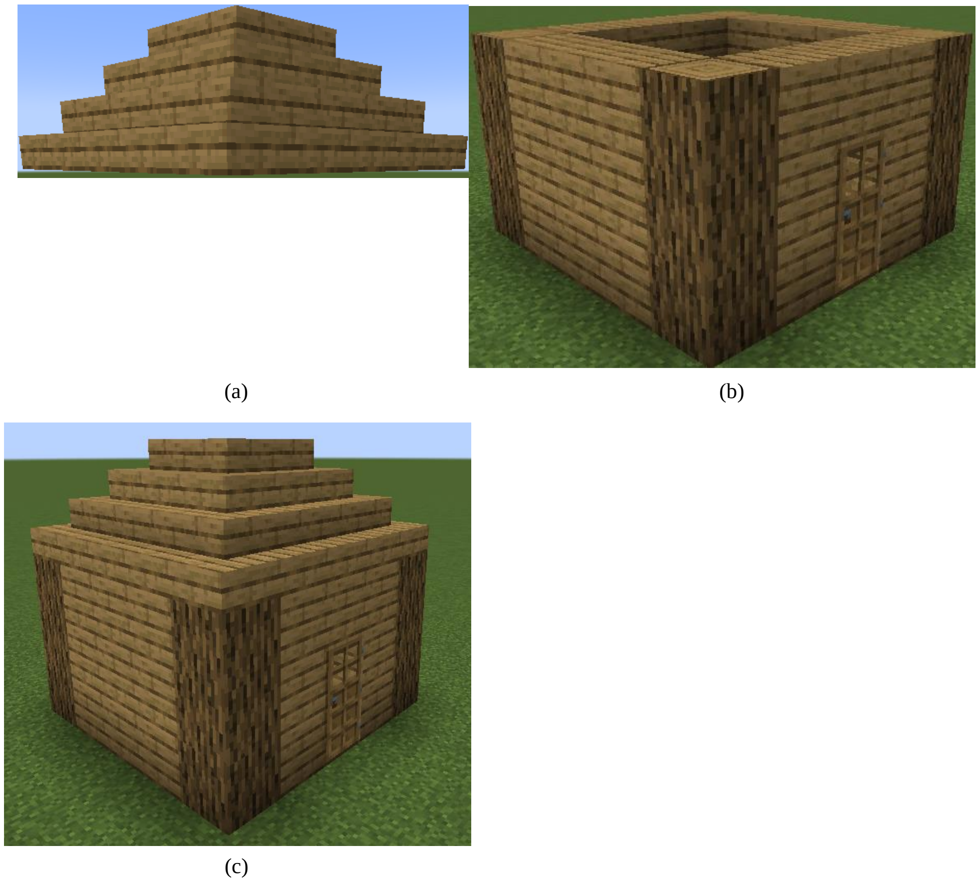


Figure 2 - (a) Example roof (b) Example house (c) Example structure created by combining (a)+(b)

House Model

The aim of the house model is to generate a 3D block for the roof to sit on. In the first design iteration, the house model simply predicted a full 3D block. The issue with this was the model had to be taught to leave a gap in the centre and 1 block for the outside wall. Just learning this took the model a long time and it was not guaranteed, which severely impacted the design of the house. Another issue was the generation speed. Producing a full 3D block was very computationally intense, especially for larger structures. Another approach took advantage of the wall being 1 block thick and empty inside. This allows for the wall of the 3D block to be mapped to a 2D tilemap. Now the model can predict the 2D tilemap and always keeps the correct house structure while being much more computationally efficient.

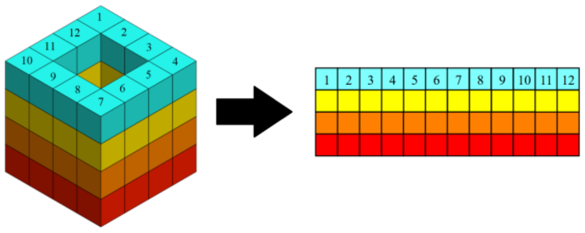


Figure 3 - 3D house block converted to 2D tilemap

Roof Model

The roof model, similar to the house model, initially predicted a 3D roof but had the same issues as the house model. The other approach to this is very similar to the house model. Since the roof should be one block thick and cover the area underneath, it can be shown as a 2D tilemap describing the height of each block in the roof. An example of a heightmap is shown in Figure 4.

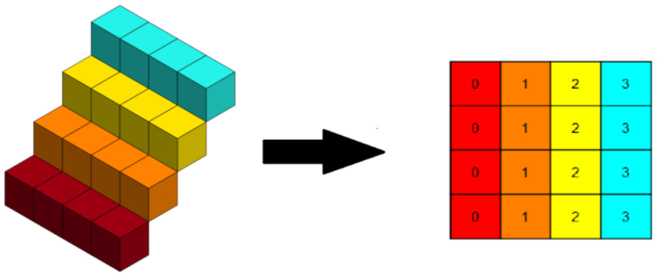


Figure 4 - Example 3D roof converted to 2D tilemap

Model hyperparameters

Hyperparameters are the parameters which control the learning process for each population, such as mutation rate and population size. One of these parameters is the input/output sizes of the model, which must be a fixed value for all individuals in a population at all times. In the first design iteration, the model predicted each block in the structure in one go. This meant the output size of the model each block used in the structure, and because the output size is fixed, the size of the structure was fixed. This massively reduced the diversity of the structures and was visually boring.

The next design iteration took a different approach. Instead of predicting all the blocks in one go, the model predicted one block at a time, denoted $\hat{y}$. This removes the limitation on the structure size, since the model can be run an unlimited number of times. This does increase the overhead as making a prediction from the model is an intense process, but it is worth it to improve the quality of the structures. Since both the house and roof model produce 2D tilemaps, they work in similar ways. An 2D tilemap is first randomized and padded by 1. To predict the first ŷ, the model is given the blocks surrounding it (denoted x_{surr}), shown in Figure 5. The output size of the model (ŷ), is each block which is possible to place. Minecraft contains 133 blocks which are suitable for the structures, and therefore is the outputs size.

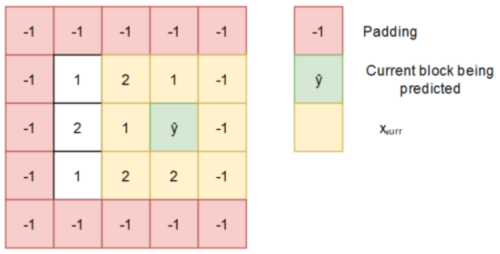


Figure 5 - Example of gathering data for model to predict ŷ

Since the models can produce structures of arbitrary size, the models need to know what size structure is being created to control the patterns, therefore the height, length and width of the structure is passed into the model as well. Also since the model has no memory, it is also given the xy coordinates of ŷ for more information about what is being generated. The house model has another input. Since the model is predicting the blocks to place, there needs to be some control over this, otherwise the model will only ever predict one block. Therefore, the house model is also given 3 seed blocks which it should use to build the structure.

Fitness Functions

A fitness function is used to evaluate the quality of a candidate solution, to help it reach a desired solution. According to (Mallawaarachchi 2017) a fitness function should be:

- **Clearly defined**: it should be easy to understand and provide meaningful insight into the performance.
- **Intuitive**: Better solutions should get a better score and visa-versa
- **Efficiently implemented**: NEAT requires many generations to evolve a good solution, so the fitness function should not be a bottleneck.
- **Sensitive**: Should be able to distinguish between slightly better and slightly worse solutions to allow a gradual movement towards a better solution

For each model the fitness is calculated by combining the novelty score and structure score (explained in more detail below). The ratio of these combinations decides the level of novelty in the population.

Structure Scoring

House Model

To start calculating a house score, the components of the desired structure have to be broken down. When deciding fitness functions it is important to limit the scope of what can be expected, there is always more detail that can be added to the scoring. The purpose of the scoring is to highlight the important parts of a structure and guide the genome towards the desired output, but leaving enough flexibility to allow for creativity. A balance between control and creativity. Each of the component scoring functions generate a value between $0 < x < 1$ and the average is used to calculate the overall structure score. The fitness functions I decided on were:

1. **A bounding wall**

An important part of a house is a complete wall with no air-gaps. While the wall cannot be structurally wrong, the model can still place airgaps which must be minimized. The score is calculated by taking the percentage of blocks that aren't air.

2. **A door**

There should be a door on ground level to enter the structure. This is the only score which is either 0 (no usable door) or 1 (usable door).

3. **Seed blocks**

The 3 seed blocks inputs are to control the types of blocks which are generated by the structure. This scores models higher which prioritises using blocks from the three seed blocks given. For each seed block which is in the top 5 blocks used, 1/3 is added to the score.

4. **Symmetry**

In almost everything considered beautiful, there is some element of symmetry. It has been well documented that humans find symmetry much more attractive and soothing to look at (Huang et al. 2018). Instead of limiting my model and forcing symmetry in one axis, I am checking both x & y axes and choosing the one which has the highest symmetry. The symmetry is calculated by comparing the percentage of equal blocks on both sides of the axis, giving a value between 0 and 1.

- Block Variance

Block variance promotes a higher variance of blocks, by scoring a structure higher if more unique blocks are used, the equation for which is below. The model gets a higher score the more blocks it uses, up to 3 blocks. Above 3 blocks the score caps at 3/4, chosen because it is better than 2 blocks (2/3) but not better than 3 blocks. This was done to prevent the the models using too many blocks and making a cluttered design, but not too few blocks.

```
int x: size(unique blocks)
if (x <= 3){
    score=x/3
}else {
    score = 3/4
}
```

The final house structure score is calculated as the average of bounding wall, door, symmetry, block variance scores.

Roof Model

Since this is a simple structure with a lower search space, it is very easy for the fitness functions to be too controlling. For example, if the scores maximize for sloped roofs, then only sloped roofs will be generated, therefore it was very important to have gentle control. The structure scores:

1. **Compliance**

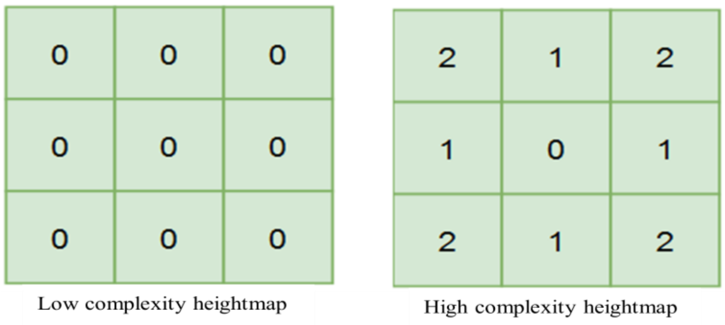
This ensures the heightmap generated fits within the limits I have set. For example, if the maximum height set is 5 blocks and the heightmap contains a height greater than 5, then the solution doesn't pass the compliance check. This fitness is either a 0 (failed the check) or 1 (passed the check).

2. **Symmetry**

Just like the house model, I am maximizing symmetry in the xy axes.

3. **Visual Complexity**

This is the fitness for quality checking and making the solution more interesting. To make the solutions more interesting I am maximizing for complexity in the surface. This is achieved by rewarding changes in height over flatness. This is accomplished by counting the ratio of changes in height of 1 vs no changes in height. I am purposely not counting changes in height of more than 1, as that will leave gaps and ruin the look of the roof. An example of low complexity vs high complexity is shown in Figure 6. I chose this solution over rewarding specific types of structure, like flat or triangle-sloped roofs, to allow for more flexibility.



0	0	0
0	0	0
0	0	0

Low complexity heightmap

2	1	2
1	0	1
2	1	2

High complexity heightmap

Figure 6 - Low complexity vs High complexity heightmap

Novelty Scoring

As previously mentioned, the novelty score is necessary to allow a population to escape any local maxima and reach a higher fitness. When finding the novelty of a genome, it needs to be compared to its closest relations. If a genomes if very different to its closest relatives, then it will have a high novelty score. An individual cannot be compared to the whole population, it has to be compared to its closest relatives. If an individual is only compared to distant relatives, which are completely different, it will always have a high novelty score. This means the novelty score will stop reflecting how the individual has changed, therefore an individual can only be compared to its closest relatives. To make this comparison, the novelty score of an individual is he average genetic distance between the k-nearest-neighbors (KNN), equation shown in Equation 2. After some testing, I found $k = \frac{2}{3} * size(population)$ gave the best comparison.

$$NoveltyScore(x) = \frac{1}{k}\sum_{i=0}^{k} d\,ist(x, \mu_i)$$

Equation 2 - Equation to calculate novelty score for an individual in a population by using KNN

Another potential issue with novelty scoring is backtracking or cycling. This can occur when a population repeatedly cycles between the same behavior space, which can repeatedly give a high novelty score, and therefore high population fitness. To a population this is an easy solution for reaching a high fitness, but does mean the population will not continue to explore the problem search space, therefore not improving. A solution to this is archiving highly novel members of the previous generation, then comparing to them when calculating novelty scores. If a population cycles, the population novelty score will be reduced and individuals who escape the cycle can improve (Salehi, Coninx, and Doncieux 2022). To choose which individuals are added to the archive, a threshold is created (α), and any individuals over that threshold are kept. From some simple testing, $\alpha = 0.85$ archived individuals at the correct rate.

City construction

Part of the aim of the project was to use procedural content generation to generate multi-structure content, e.g. a city. To generate a city using a trained house and roof populations, individuals are randomly selected to place a house and roof. By using all individuals from the population, the novelty of a population can come across in the constructions. The generated houses start off at the same sizes, but every time a building is placed there is a 5% chance for the sizes to be adjusted. This gives a gradually chances the dimensions of the houses, instead of having randomly sized buildings next to each other, mimicking real city.

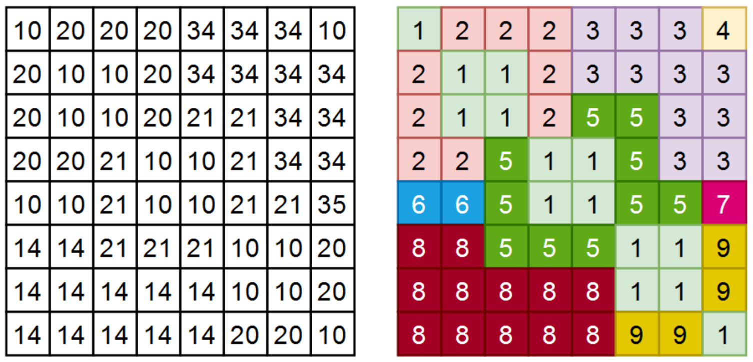
A core part of the project is maximizing diversity while maintaining the structural form, therefore this needs to be measured across the city. 1. Structure Score distribution The structure score is my measurement for the structural form of a building. If all the generated structures in a city have a low structure score then the constructions will be poorly made and will look worse.

1. **Block Distribution**

The visual quality of the structures are broken down into two metrics, colors and patterns. The color distribution is easier to measure because each block has roughly a different color. Therefore if the structure has a high distribution in blocks used, the color distribution will be higher and therefore will be more diverse.

2. **Pattern Measurement**

Pattern measurement is how often a structure design is repeated .This is harder to measure because the buildings aren’t all the same dimensions and are each made of different blocks. Since patterns can feasibly only be compared with structures of the same size, they are grouped by the dimensions and a pairwise comparison is made within the groups. Next each structure is passed through the skimage labelling function Figure 7. This function labels connectivity between the elements of an array. An element is connected to another element if it is a neighbor and they have the same value. An example of connectivity is shown in @(**???**). By labelling the connected regions in the image it is much easier to identify patterns between structure, regardless if different blocks are used. The labelled structures are then directly compared and the percentage similarity is used as the pattern similarity value.



10	20	20	20	34	34	34	10
20	10	10	20	34	34	34	34
20	10	10	20	21	21	34	34
20	20	21	10	10	21	34	34
10	10	21	10	10	21	21	35
14	14	21	21	21	10	10	20
14	14	14	14	14	10	10	20
14	14	14	14	14	20	20	10

Randomised pattern

1	2	2	2	3	3	3	4
2	1	1	2	3	3	3	3
2	1	1	2	5	5	3	3
2	2	5	1	1	5	3	3
6	6	5	1	1	5	5	7
8	8	5	5	5	1	1	9
8	8	8	8	8	1	1	9
8	8	8	8	8	9	9	1

Labelled pattern

Figure 7 - Randomized pattern being labelled using skimage labelling function

3. Generation Time

One of the drawbacks of PCG, identified in the introduction, was the generation speeds. The immersion of a game can be broken if the user must wait too long for the world to be generated, therefore the generation speed needs to be minimized.

Evaluation

Experimentation

The aim of the experimentation is to understand the effects of different levels of novelty in populations for building structures. When an individual's fitness is calculated, their structure and novelty scores are calculated and the ratio of these are used for the fitness. The ratio of structure:novelty determines the level of novelty in the population. To experiment with the novelty, different populations will be trained, each with a different level of novelty. The structure scores throughout training will be collected and compared to show the effects of novelty throughout the training process. For each level of novelty a house and roof model will be trained twice. Since the models are inherently random, running the generations multiple times was necessary to show a population wasn't an anomaly. The levels of novelty that will be compared are: no novelty (control), low novelty (1:4 novelty:structure), high novelty (4:1 novelty:structure), and full novelty (1:0 novelty:structure).

Each house model trained had a population of 20 and each roof model trained had a population of 100. Having a larger population increases the chances of a positive mutation, but also increases the hardware requirements. The neat-python library does not have GPU support and can struggle at larger populations, so was limited to those population sizes. Each model was trained for 2000 generations and was then trained for longer if the population was still showing improvement. The training was then stopped when the models have not improved in the last 50 generations, because there should be some improvement in that time. Unless the models reaches the the highest possible score (1.0), they can always train for longer because there is a chance they might improve more, but there has to be a cut off.

Training Comparisons

Figure 8 shows an overview of the training results. The graph shows the maximum population average over a 20 generation window. This means every generation, the previous 20 generations are taken, and the average is taken over that window, equation shown in Equation 3. As shown later, this is necessary because the graphs are quite noisy. If just only max value was taken, without the window, all the averages are very similar and don't reflect the state of the population. The graph shows that there is quite a difference between runs, but overall, the results line up with what was expected, with the high novelty model regularly having the highest performance and the full novelty model having the lowest performance.

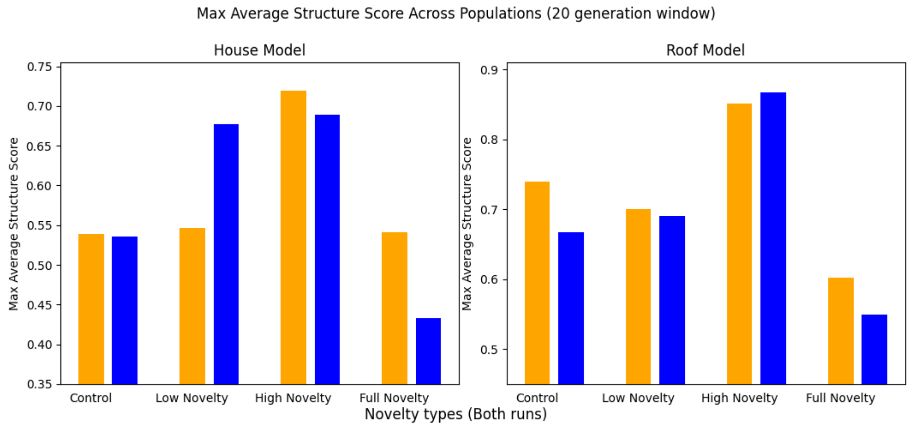


Figure 8 - Graphs for house and roof model, comparing maximum average structure score achieved by different novelty populations, and different runs.

$$AverageMaxStructureScore = Max(\{Average(x_{i-20}, \ldots, x_i)\}) \qquad \forall i \in \mathbb{Z}, 20 \leq i \leq N$$

Equation 3 - Equation to get maximum average score from a list of scores (x)

Control Model

(Figure 9) The control model shows a very typical learning process without novelty, where there is a region of rapid improvement at the start as the model can easily make improvement, but then reaches a local maxima, where the average score stays. Without the novelty mechanic, the model is stuck there and cannot improve. Since this has happened over all 4 graphs, we can be confident this would happen a majority of the time. The success of the control model does vary and can sometimes surpass models with novelty, but it is very dependant on when it reaches the first local maxima, and would not improve with more generations, whereas a model with novelty might.

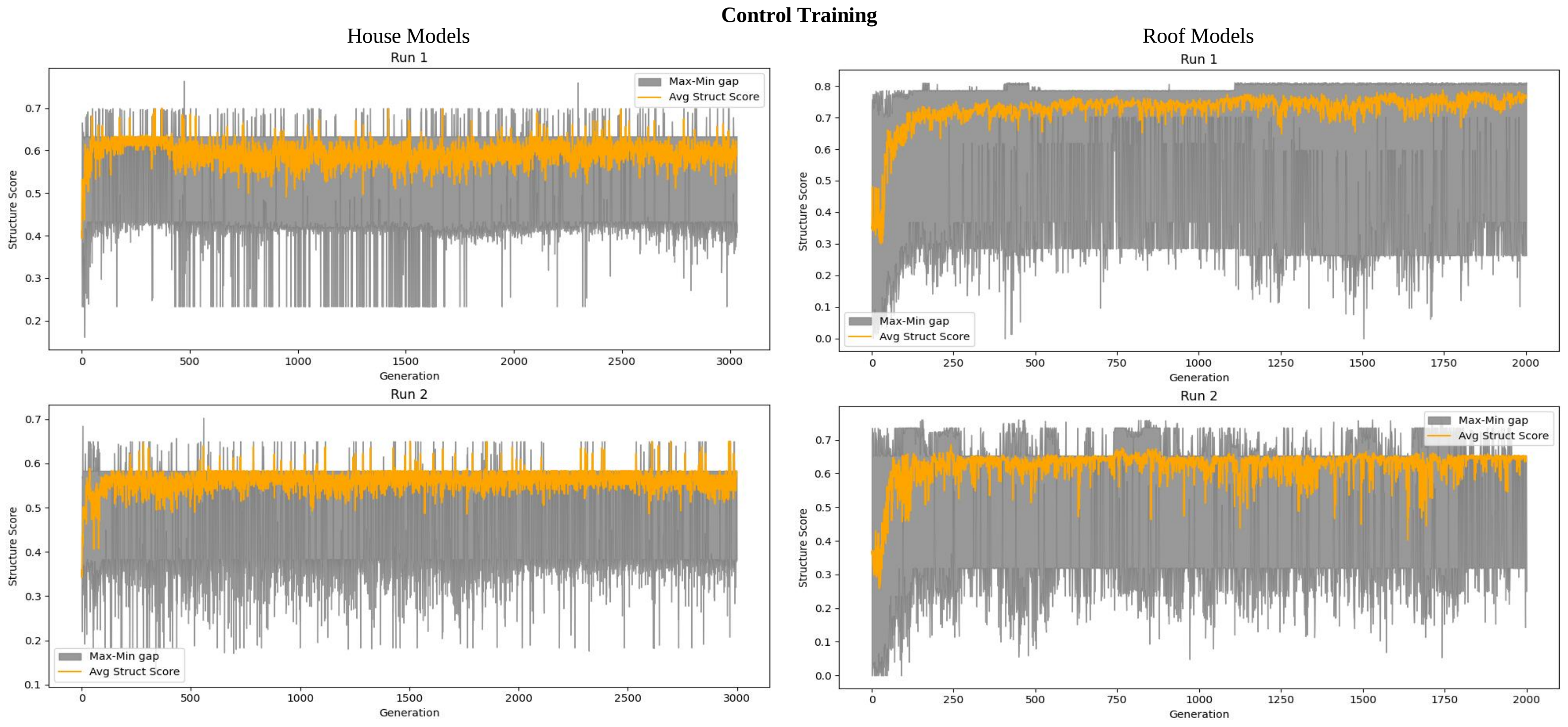


Figure 9 - All runs of training for control models (house and roof)

Low Novelty Model

(Figure 10) The low novelty models show lots of similarities with the control models, which is to be expected. There is more variation where the model has tried to escape a local maxima, especially evident in the run 2 of the house model around generation 250 - 750, where there is a significant dip with a large amount of variation. This period also has a large max-min gap (shown in grey). The large gap between the lines is evidence of novelty in the population, where the best and worse models have diverged. We should expect this max-min gap to appear more in the roof model as the population 5x larger and can spread out much easier.

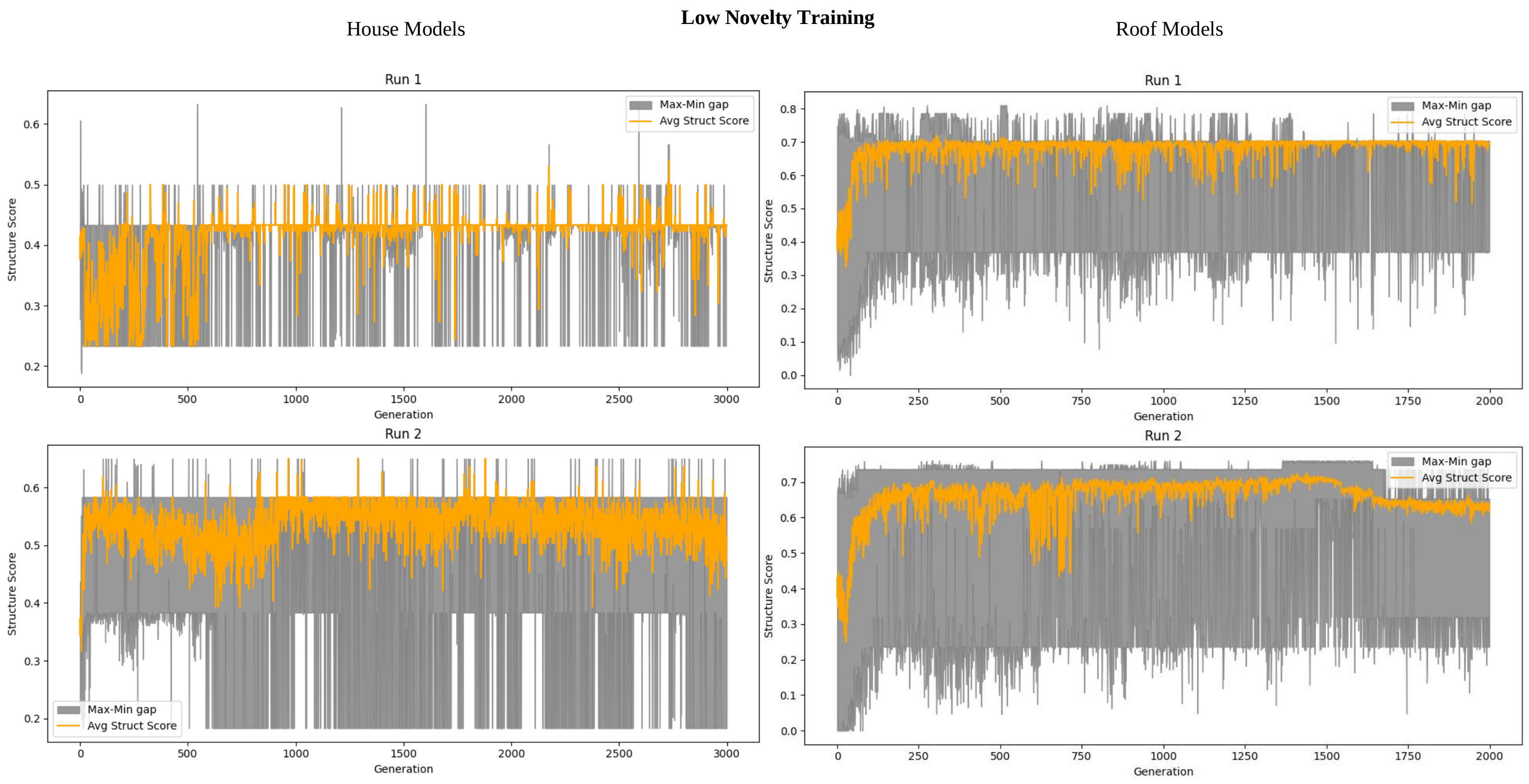


Figure 10 - All runs of training for low novelty models (house and roof)

High novelty Model

(Figure 11) The high novelty house model clearly shows a lots of activity during training, whereas the roof model shows much less. My hypothesis is that since the roof is much simpler and a smaller search space, the model either found a global maxima or a local maxima which it couldn't escape from. What the graph does show is a large max-min gap which is again evident of novelty. On the other hand, the house model clearly shows how novelty has caused many dips in the structure score. Especially evident around generation 400 on run 1, where there is a significant dip, rise in the middle, and dip again.

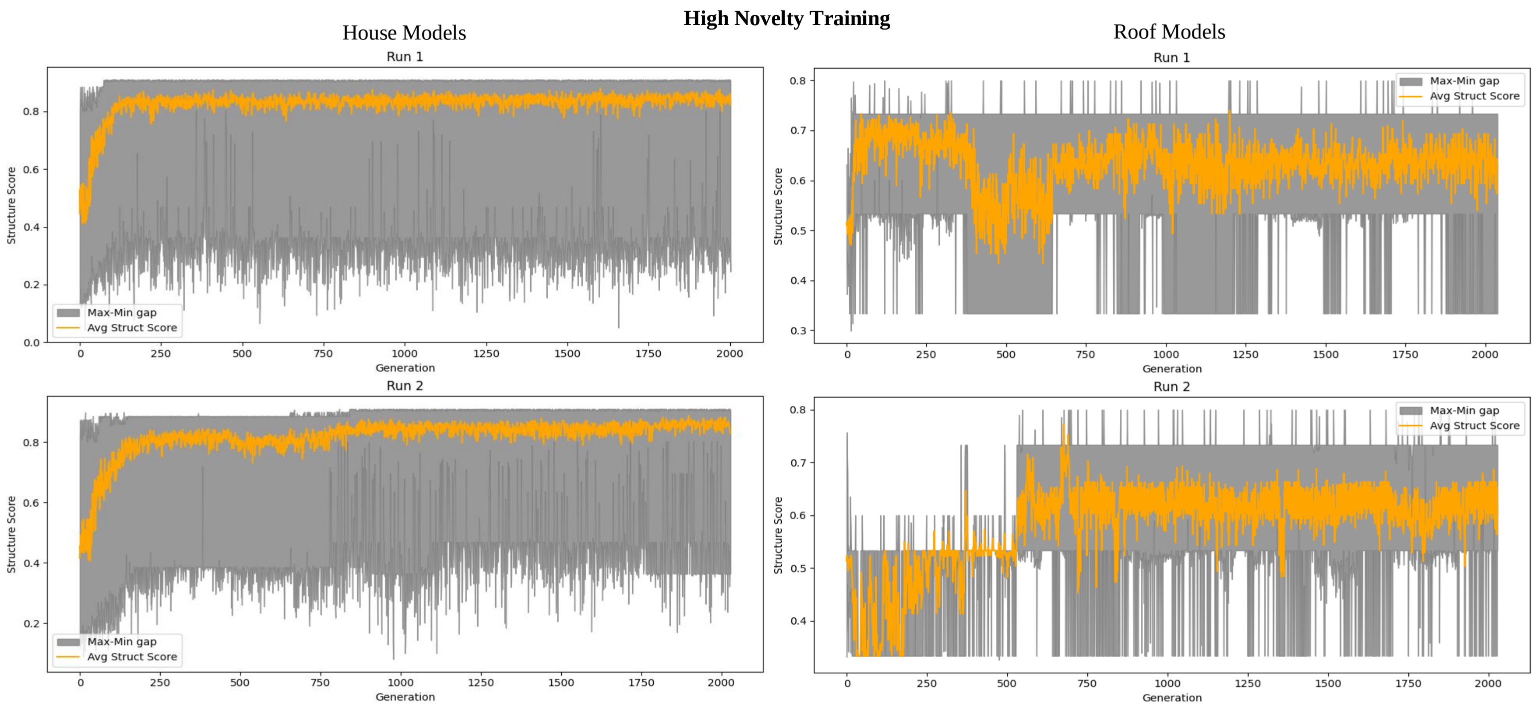


Figure 11 - All runs of training for high novelty models (house and roof)

Full Novelty Model

(Figure 12) Because these models had no structure score feedback, only novelty score, there is little progression through the training. These models are randomly hoping to stumble across the correct solution and there are times, like in run 1 of the house model around generation 1750, where the model has found a decent solution. Unfortunately, since the model didn't know it had improved, it quickly evolved away from that solution instead of improving on it. Run 2 of the roof model does appear to show the model retaining the score for a significant period. This could be because the model was evolving changes which had no effect on the score and therefore didn't reduce/improve the score.

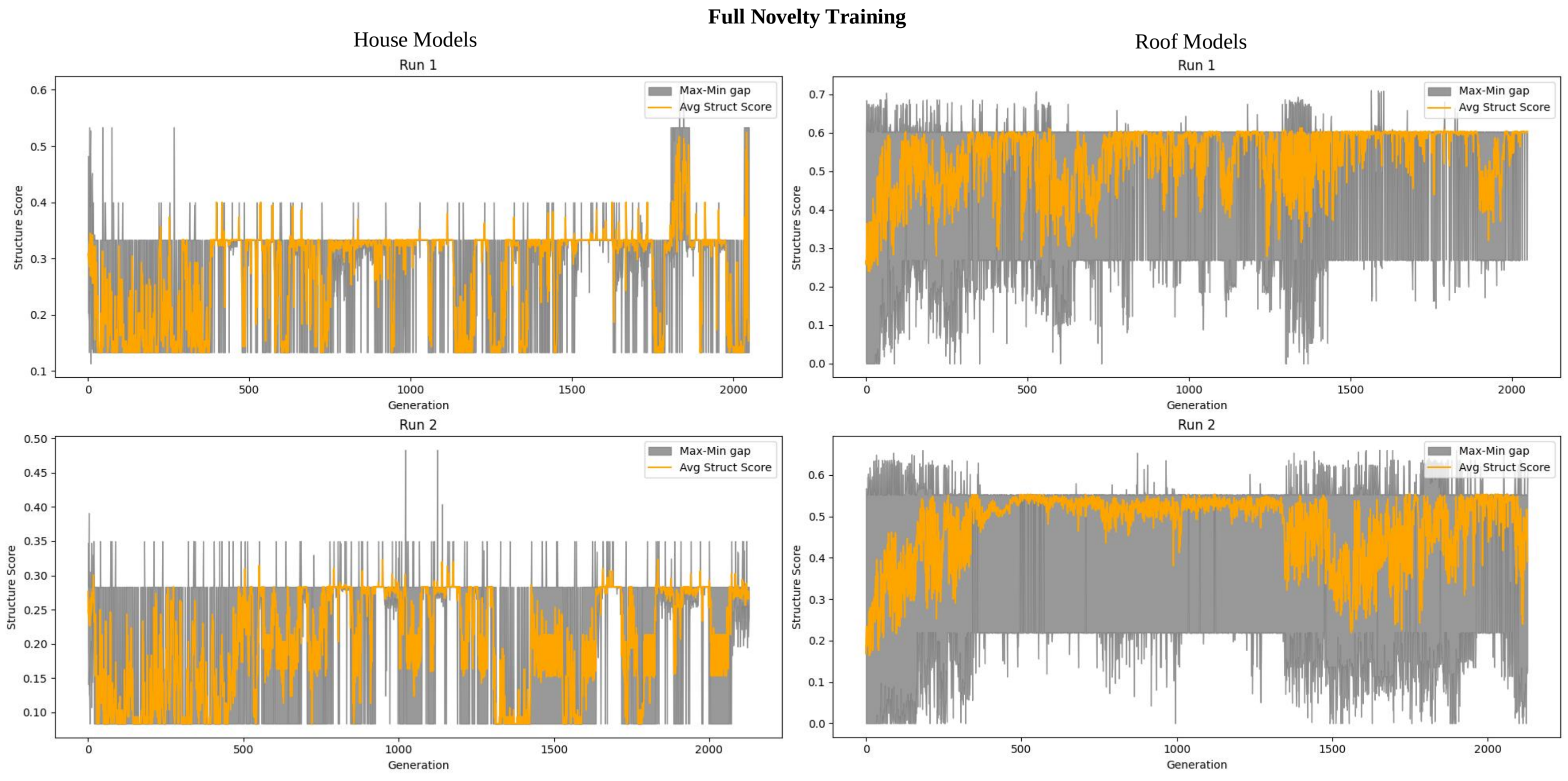


Figure 12 - All runs of training for full novelty models (house and roof)

Multi-structure Comparisons

To populate the datasets for comparison, 20,000 structures are generated using each novelty type. Since some of the metrics rely on grouping building sizes, it is important there is a large number of structures to get an accurate comparison. ### Structure Score Distribution

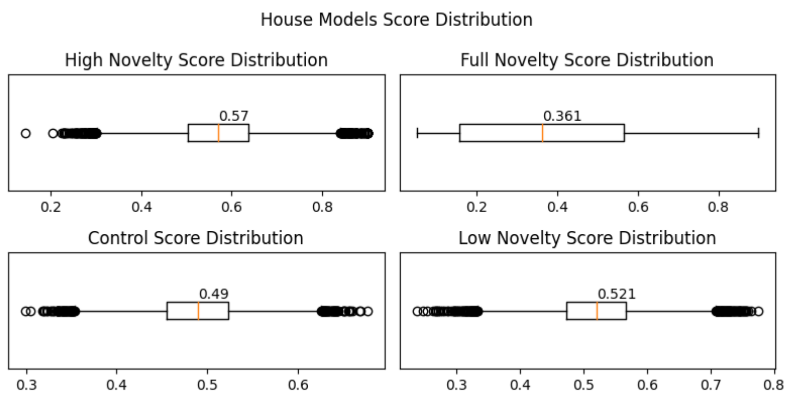


Figure 13 - Score distribution of generated houses for different models

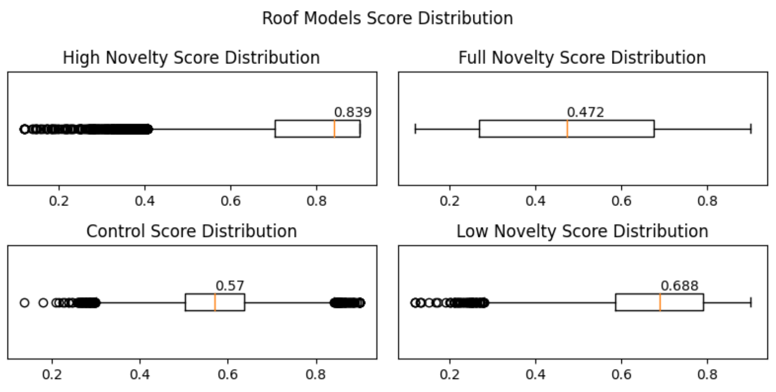


Figure 14 - Score distribution of generated roofs for different models

Block Distribution

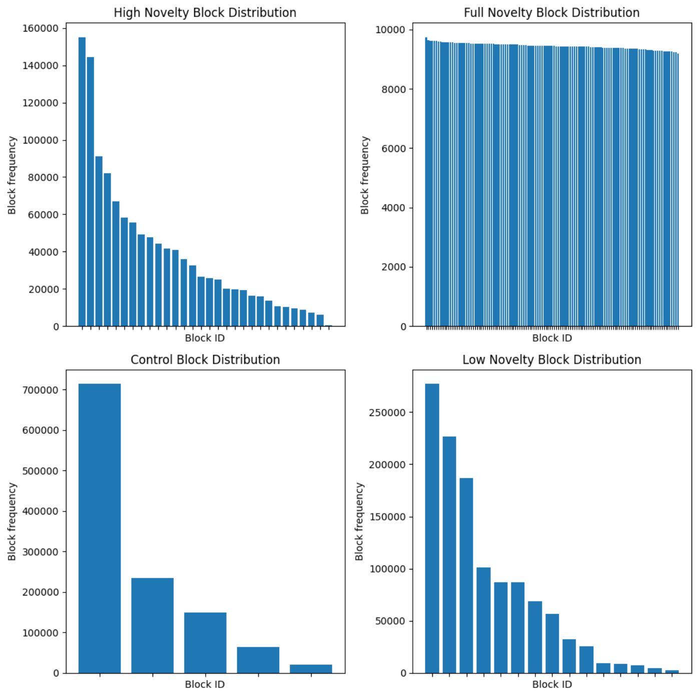


Figure 15 - Block distribution of generated structures for different models

Pattern Variety

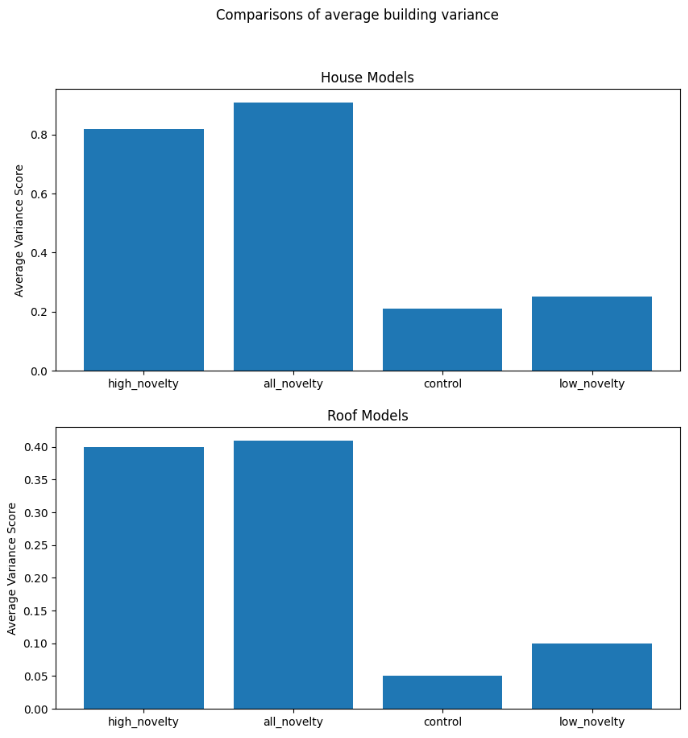


Figure 16 - Variance in building patterns for different models

Generation Time

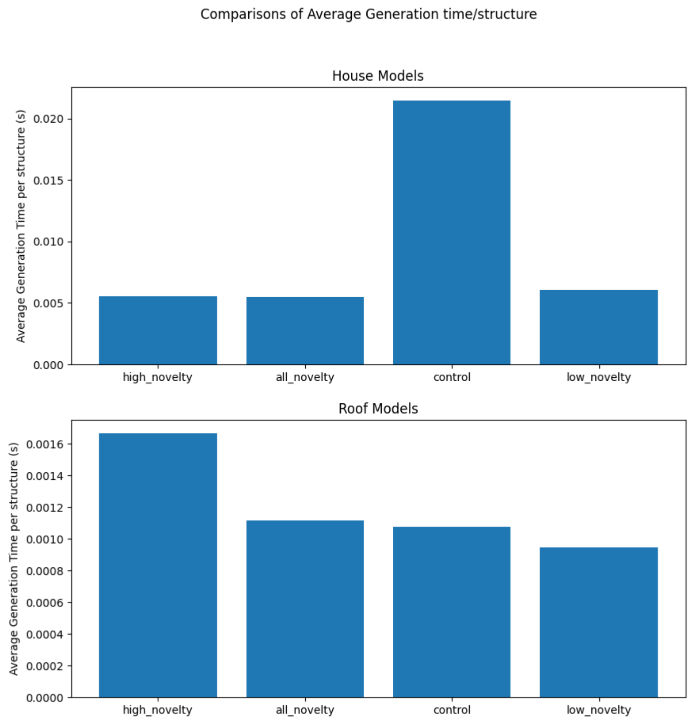


Figure 17 - Generation times for roofs and houses for different models

Future Work

References

Beukman, Michael, Manuel Fokam, Marcel Kruger, Guy Axelrod, Muhammad Nasir, Branden Ingram, Benjamin Rosman, and Steven James. 2023. "Hierarchically Composing Level Generators for the Creation of Complex Structures." http://arxiv.org/abs/2302.01561.

Carlson, Patrick. 2013. "Arma 3 Map Might Be Bigger Than You Ever Imagined." *Pcgamer*. PC Gamer. https://www.pcgamer.com/arma-3-map-might-be-bigger-than-you-ever-imagined/.

Games, Hello. 2016. "No Man's Sky." *No Man's Sky*. https://www.nomanssky.com/.

Godefroid, Patrice, and Sarfraz Khurshid. 2002. "Exploring Very Large State Spaces Using Genetic Algorithms." In *Tools and Algorithms for the Construction and Analysis of Systems*, 266–80. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-46002-0_19.

Grbic, Djordje, Rasmus Berg Palm, Elias Najarro, Claire Glanois, and Sebastian Risi. 2020. "EvoCraft: A New Challenge for Open-Endedness." http://arxiv.org/abs/2012.04751.

Greenwald, Noah. 2009. *The Elements of Biodiversity*. Centre for Biological Diversity. https://www.biologicaldiversity.org/programs/biodiversity/elements_of_biodiversity/.

Heidenreich, Hunter. 2019. “Neat: An Awesome Approach to Neuroevolution.” *Towards Data Science*. Medium. https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f.

Huang, Yi, Xiaodi Xue, Elizabeth Spelke, Lijie Huang, Wenwen Zheng, and Kaiping Peng. 2018. “The Aesthetic Preference for Symmetry Dissociates from Early-Emerging Attention to Symmetry.” *Scientific Reports* 8 (1). https://doi.org/10.1038/s41598-018-24558-x.

Hui, Jonathan. 2018. “Why It Is so Hard to Train Generative Adversarial Networks!” Medium. https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b.

Interactive, Bohemia. 2012. https://arma3.com/.

Joy, Ashwin. 2019. “Pros and Cons of Reinforcement Learning.” *Pythonista Planet*. https://pythonistaplanet.com/pros-and-cons-of-reinforcement-learning/.

Koster, Raph. 2013. *A Theory of Fun for Game Design*. O’Reilly Media.

Mallawaarachchi, Vijini. 2017. “How to Define a Fitness Function in a Genetic Algorithm?” *Towards Data Science*. Medium. https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4.

Newhouse, Alex. 2016. “Making Sense of No Man’s Sky’s Massive Universe.” GameSpot. https://www.gamespot.com/articles/making-sense-of-no-mans-skys-massive-universe/1100-6441344/.

OpenAI. 2022. “ChatGPT Chat.” *ChatGPT*. https://chat.openai.com/.

Park, Minyoung, Minhyeok Lee, and Sungwook Yu. 2022. “HRGAN: A Generative Adversarial Network Producing Higher-Resolution Images Than Training Sets.” *Sensors* 22 (4): 1435. https://doi.org/10.3390/s22041435.

Persson, Markus. 2023. “Welcome to the Minecraft Official Site.” *Minecraft.net*. Mojang Studios. https://www.minecraft.net/en-us.

Ramesh, Aditya, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. “Hierarchical Text-Conditional Image Generation with Clip Latents.” http://arxiv.org/abs/2204.06125.

real-itu. 2021. “Starter Code for Simple Block Evolution Loop.” *Simple Minecraft Evolver*. Robotics Evolution; Art Lab. https://github.com/real-itu/simple_minecraft_evolver.

Salehi, Achkan, Alexandre Coninx, and Stephane Doncieux. 2022. “Geodesics, Non-Linearities and the Archive of Novelty Search.” http://arxiv.org/abs/2205.03162.

Stanley, Kenneth O., and Risto Miikkulainen. 2002. “Evolving Neural Networks Through Augmenting Topologies.” *Evolutionary Computation* 10 (2): 99–127. https://doi.org/10.1162/106365602320169811.

Steam. 2023. “Steam Hardware and Software Survey 2023.” Valve Corperation. https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam.

Summerville, Adam, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. "Procedural Content Generation via Machine Learning (Pcgml)." http://arxiv.org/abs/1702.00539.

"The Quest for Ai Creativity." 2015. *IBM Cognitive - What's Next for AI*. IBM. https://www.ibm.com/watson/advantage-reports/future-of-artificial-intelligence/ai-creativity.html.

UserBenchmark. 2023. "Intel Uhd Graphics 630 (Desktop Coffee Lake I5 I7) Vs Nvidia Rtx 3060-Ti." https://gpu.userbenchmark.com/Compare/Nvidia-RTX-3060-Ti-vs-Intel-UHD-Graphics-630-Desktop-Coffee-Lake-i5-i7/4090vsm356797.

Whitworth, Spencer. 2021. "How Big Is a Minecraft World?" *Sports News*. Sportskeeda. https://www.sportskeeda.com/minecraft/how-big-minecraft-world.

2010. *Bukkit*. Mojang. https://dev.bukkit.org/.

Appendix

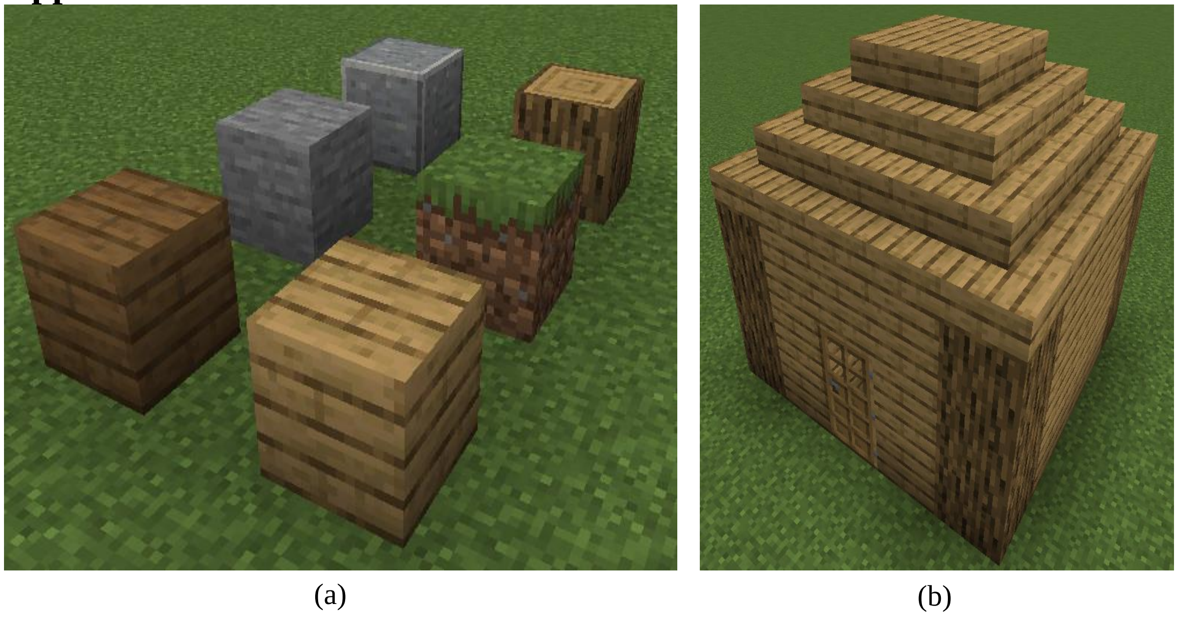
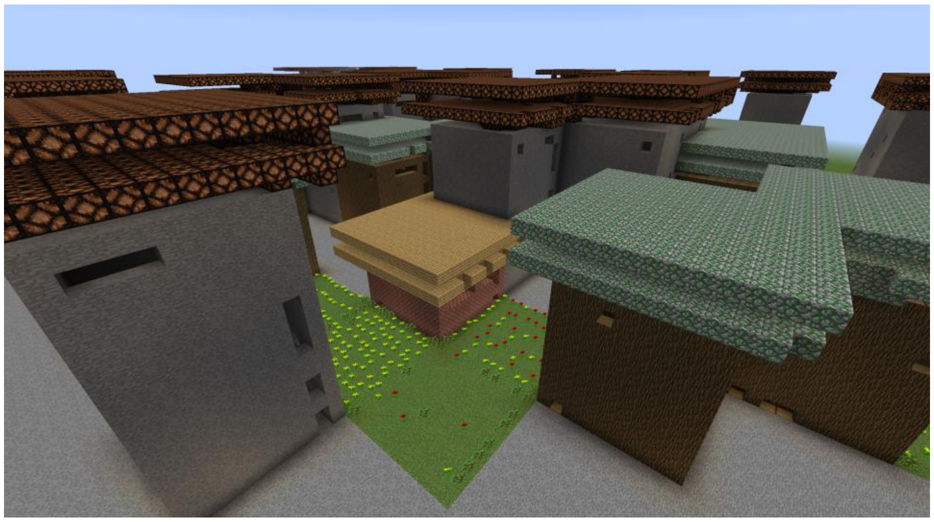
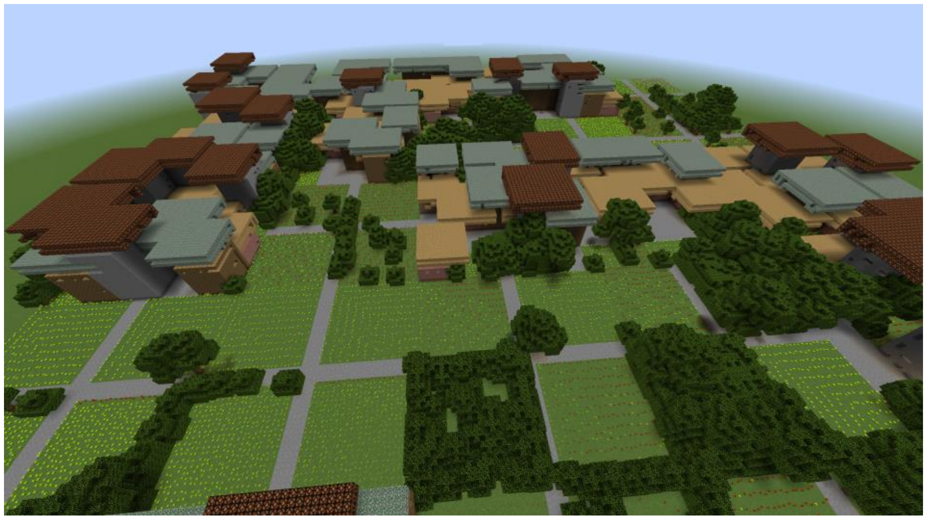


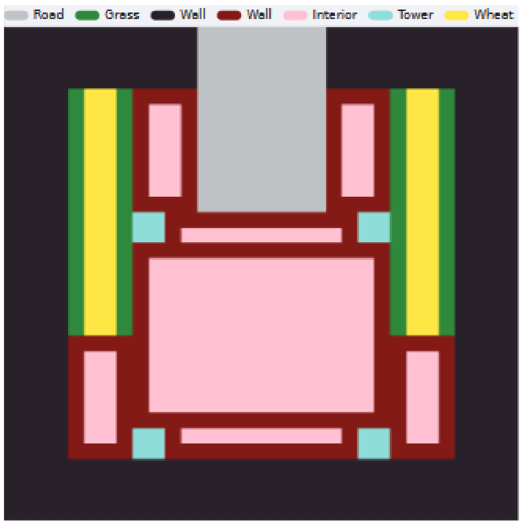
Figure 18 - (a) Sample of blocks (voxels) from Minecraft (b) Example structure built from blocks.



(a)



(b)



(c)

Figure 19 - Examples from EvoCraft PGCNN (Beukman et al. 2023). (a) Generated house, roof, garden inside a town (b) Birdseye view of a procedurally generated town (c) 2D tilemap plan for a town