Generation of diverse Minecraft structures with evolutionary algorithms

Ben Hutchings

May 2023

School of Computing Science, Newcastle University, UK

Abstract

This dissertation aims to experiment with novelty search in NEAT (NeuroEvolutiosn of Augmenting Topologies) models for procedural content generation of diverse structures in Minecraft.

Keywords: NEAT, Novelty Search, Minecraft, PCG, EvoCraft

Table of Contents

1. 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 (Hello Games, 2016), a universe exploration game, which can generate 2^{64} (18×10^{18}) worlds, each ~203 km^2[1]. Comparing this to one the biggest non-generated games, ARMA 3[2] which has a map size of 270 km^2[3], shows how much more content can be generated by PCG.

One approach to PCG is ML-PCG (Machine Learning PCG), but this comes with some problems. Unlike most other PCG content, like art or music, games have another requirement on top of artistic quality, playability. A small difference in music rarely affects the overall quality, but a small difference in a game level might lead to it being unplayable. For example, a gap is slightly too big for a player to cross, or a maze has no path through, which leads to a game level being useless. Since PCG can generate infinite content, it cannot be manually quality checked, so this must be fundamental to the generation[4].

Another issue with some ML-PCG methods is dataset bias. Many machine learning models rely on training data to improve a model. This means the quality and diversity of the dataset can significantly impact the generated content and unwanted biases can come through. Unlike other content generation, like music or art, there is no big or diverse enough dataset readily available for game level generation, which is a significant handicap [4].

While the issues with ML-PCG do rule out some methods, there are others which are suitable. NEAT [5] is a search-based algorithm which aims to find the best neural network topology for a problem. It does this using a genetic algorithm which mimics natural selection to “evolve” the best network. The algorithm starts with a population of very basic randomized neural networks. The algorithm then uses a fitness function to score each network’s performance. The best networks then mutated, which is where nodes are randomly added or weights between nodes are randomly changed. The hope is that through random chance the network will “evolve” and improve. Because this solution is evolved, rather than learnt from training data, the handicaps previously identified are less significant. Since there is no dataset, there can be no dataset bias, but this can still be introduced through fitness functions. Problems with playability can also be overcome by promoting individuals which produce playable candidate solutions.

However, there is a problem with NEAT which is common to many genetic algorithms and objective-based searches, deception. Deception is when a seemingly successful individuals falls into a local maximum, unable to reach a global maximum and breaking the evolution algorithm. A solution to this is novelty search, which rewards individuals for being different to others in the population, adding a constant pressure to do something new. By encouraging individuals to change, diversity is also encouraged. Now when an individual hits a local maximum, instead of leading the whole population with it, there is more likely to be another which does get into a local maximum, improving the odds of reaching the global maximum[6].

In this dissertation I will be experimenting with novelty search using the to generate diverse structures in Minecraft [7]. To do this I will be using the EvoCraft competition[8] as a framework.

1.1 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.

2. **Background**

2.1 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 30. Minecraft uses PCG and world seeds to create a unique world which is 3.6 billion blocks^2[9], 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. For interactions between Python and Minecraft world I will be using a Minecraft server with a the RaspberryJuice plugin[10] installed and the McPi Python library[11] to communicate with it. 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.

2.1.1 The EvoCraft Challenge

To investigate genetic algorithms for PCG, I am using the EvoCraft Challenge as a framework to base the project on. The EvoCraft challenge brief is to create an open-ended algorithm which can create novel and increasingly complex structures in Minecraft. These algorithms must 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.

Simple Minecraft Evolver

[12] The simple Minecraft evolver is basic implementation of NEAT 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. The fitness of a candidate solution is calculated by the distance to the gold block. This simple implementation gives a good practical demonstration of the evolution process, which is useful for understanding more complex implementations.

Evocraft PCGNN

Beukman et al.[13] were 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 put towns together to create a city, which could grow endlessly. Example of this process shown in Figure 31.

2.2 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 $y = f(\sum_{i=0}^{n_{prev}} w_i\, x_i + b)$, where W_i is the weight to a previous node, X_i is the value of the node, and b is a bias.

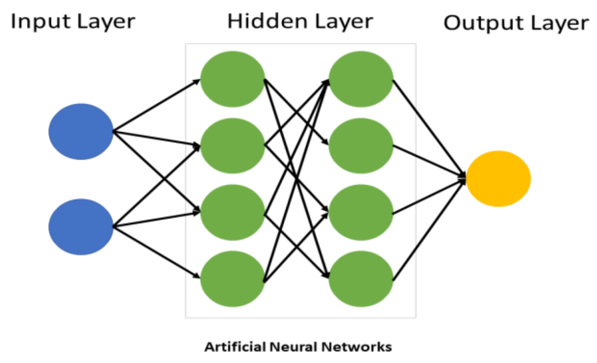


Figure 1 - Example Neural Network Layers

To make a prediction, the input nodes values are set as the user-defined input 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 are the prediction. A tradsitional neural network can learn by adjusting the weights and bias in the neurons.

2.3 NEAT + Novelty Search

As mentioned in the introduction, NEAT is an algorithm which combines the principles of genetic algorithms and neural networks. The aim is to evolve a neural network topology which best solves a problem. To evolve the neural networks, the algorithms represent them as a genome, an example is shown in Figure 2**Error! Reference source not found.**. The initial population starts very simple, as just connections between input and output nodes, without hidden nodes, with the weights randomized. Each individual then has its fitness calculated by a fitness function. The fitness function gives a score, between 0 and 1, which scores how close the individuals candidate solution is to the true solution. Once scored, the top individuals continue to the next generation and the others are removed.

To evolve the population, the top individuals are used to create the next generation, which ensures the positive genes carry forward. During the evolution process, NEAT uses a combination of mutation and crossover operations. Mutation simply randomly adds nodes and/or randomly changes the weights. Crossover combines the genes from two parent genomes to create an offspring genome. This is not performed blindly as there are, as the original NEAT paper[5] calls, “competing conventions” which become an issue. Genomes with competing conventions are genomes which have evolved separately and should not be combined. Combining these would almost never be a positive evolution. To work around this, the genome has historical markers which track genes across generations. When two genomes are crossed over, the historical markers are used to match genes to ensure compatibility.

As previously mentioned, a problem with the NEAT algorithm is lack of diversity within a population. This usually manifests due to the algorithm taking the individuals with the top fitness scores, without considering the diversity. This tends to benefit in the short term but can severely reduce diversity in the long term. This is where novelty search can be used. Novelty search looks at the genotype distance of an individual to its neighbours and gives a higher novelty score if the distance is further. The genotype distance measures how different two genome are to each other based on their genes. The novelty score then contributes to the overall fitness of the individual, which allows for individuals which are not performing well but are diverse. This promotes populations to spread out over the search space and improve the odds of reaching the global maximum.

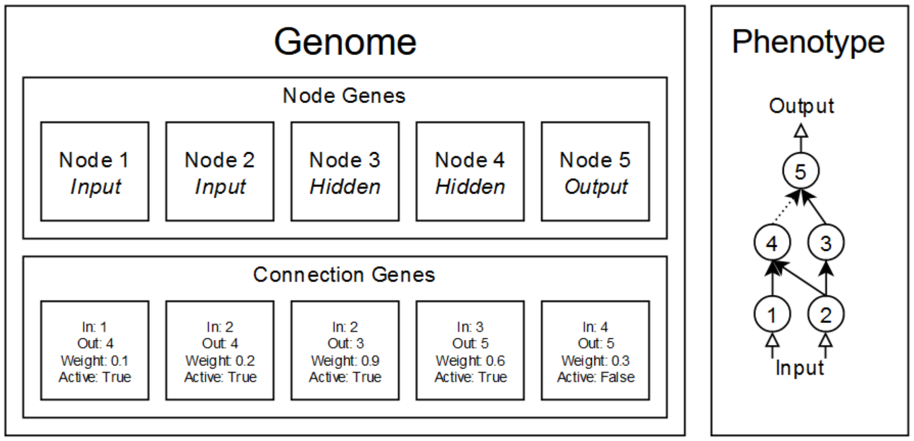

Genome
Node Genes
Node 1 Input
Node 2 Input
Node 3 Hidden
Node 4 Hidden
Node 5 Output
Connection Genes
In: 1 Out: 4 Weight: 0.1 Active: True
In: 2 Out: 4 Weight: 0.2 Active: True
In: 2 Out: 3 Weight: 0.9 Active: True
In: 3 Out: 5 Weight: 0.6 Active: True
In: 4 Out: 5 Weight: 0.3 Active: False
Phenotype
Output
5
4
3
1
2
Input

Figure 2 - Genotype (containing node and connection genes) and corresponding phenotype (neural network)

2.4 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 an environment, and will either be rewarded, if the action was a positive action, and punished, if the action was negative. The eventual goal for an 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, learning 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 maximum 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 maximum, it will have to unlearn that entire strategy and come up with a new one for it to reach a global maximum. NEAT on the other hand, contains many individuals in a population, each of which can explore their own routes (promoted through novelty search). Each route that ends in a local maximum would be killed in favor of a route which produces a global maximum, therefore having a higher likelihood of reaching the global maxima [14]. While the RL model approach has the potential to be effective in this project, the properties of NEAT make it more desirable.

2.5 General Adversarial Networks

A General Adversarial Network (GAN) is a 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 [15].

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. The discriminator can also be used to detect 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 [16]. This is not a problem for creative works as some minor variation has no effect on the quality of the output but does influence 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.

3. Implementation

I have chosen to implement this project in Python. Python already has a Minecraft API which allows for easy control over a Minecraft world. Python also has a wide array of data science libraries, including python-NEAT, which makes development much smoother. Finally, the other EvoCraft entries, including [13], are written in Python. This allows me to use them as reference to help my development.

The aim of the project is to procedurally create diverse Minecraft structures. Taking a note from *Beukman et al.* [13], the structure generation is broken down into 2 parts, one generating the base of the house and another generating the roof, an example shown in Figure 3. 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.

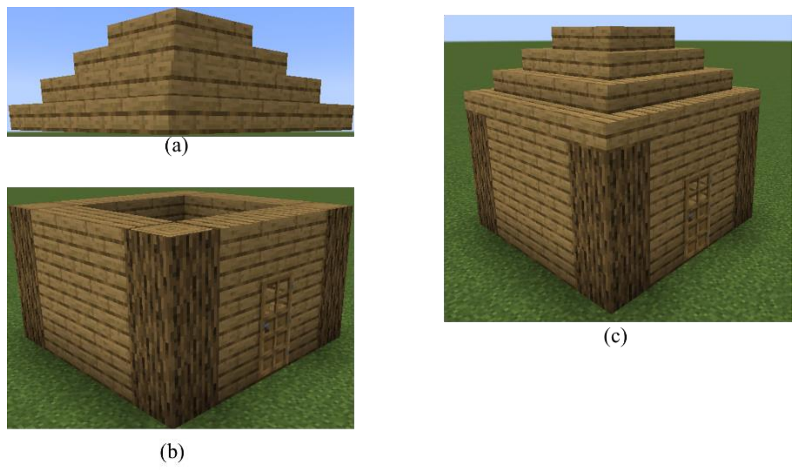


Figure 3 – Hand-created Minecraft structure (a) the roof (b) the house (c) full structure made by combining (a)+(b)

3.1 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 center 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, example shown in Figure 4. Now the model can predict the 2D tilemap and always keeps the correct house structure while being much more computationally efficient.

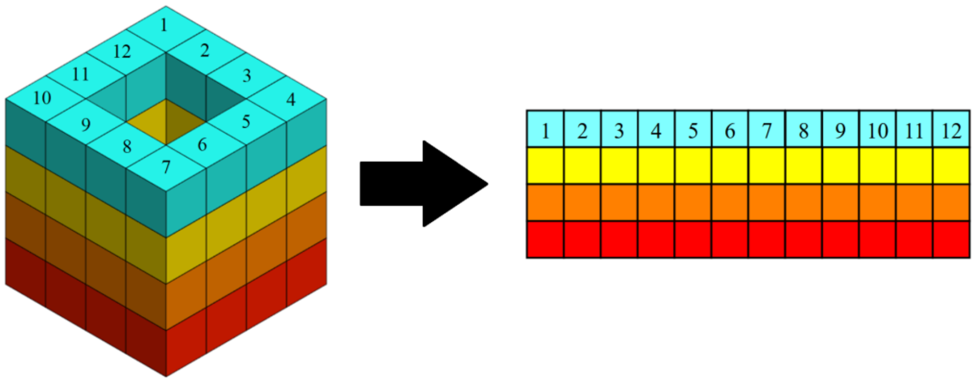


Figure 4 - 3D house block converted to 2D tilemap.

3.2 Roof Model

The roof model, like 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 5.

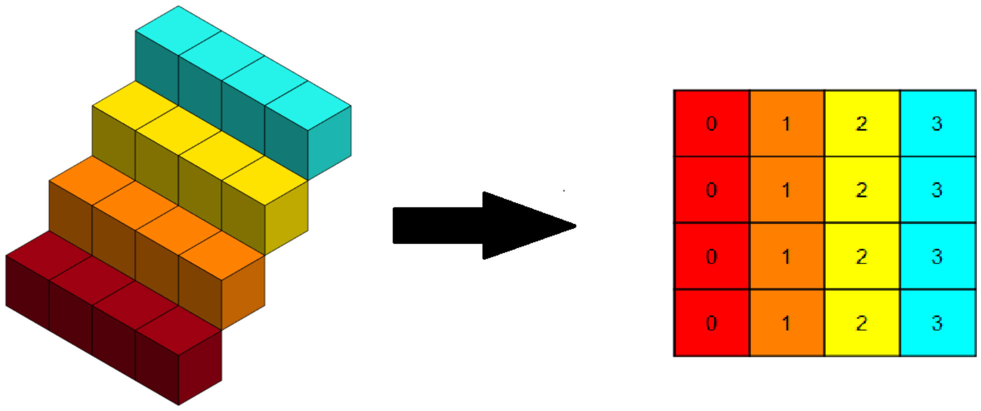


Figure 5 - Example 3D roof converted to 2D heightmap.

3.3 Model hyperparameters

Hyperparameters are the parameters which control the learning process for each model, 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 the whole structure in one go. This meant the output size of the model was the number of blocks 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 ŷ. 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. A 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 6. 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 output size.

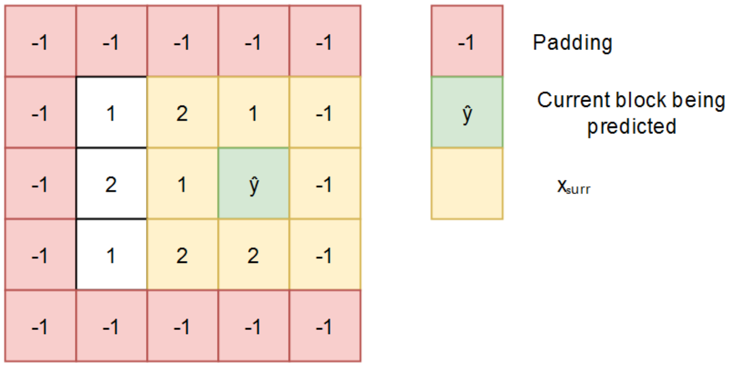


Figure 6 - Example of gathering data for model to predict ŷ.

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.

3.4 Fitness Functions

A fitness function is used to evaluate the quality of a candidate solution, to help it reach a desired solution. According to [17] 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 weighing the novelty score and structure score (explained in more detail below). The weights of novelty to structure score decides the level of novelty in the population.

3.4.1 Structure Scoring

At the end of every generation, each individual needs to be evaluated and therefore a structure needs to be generated using the individual. The aim of the individual is to be able to create a structure of any size, not a fixed size. Therefore, when an individual is evaluated the dimensions of the structure are randomized. If the dimensions were not randomized, the model would only perfect that size of structure. In an ideal world, each individual would be tested on a wide range of structure generations, but this would increase the training time massively, making it infeasible.

3.4.1.1 House Model

To start calculating a house score, the components of the desired structure must 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 while leaving enough flexibility to allow for creativity. A balance between control and creativity. Each of the component scoring functions generates a value between $0 \leq x \leq 1$ and the average is used to calculate the overall structure score. The fitness functions I decided on were:

A bounding wall

An important part of a house is a complete wall with no airgaps. 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 are not air.

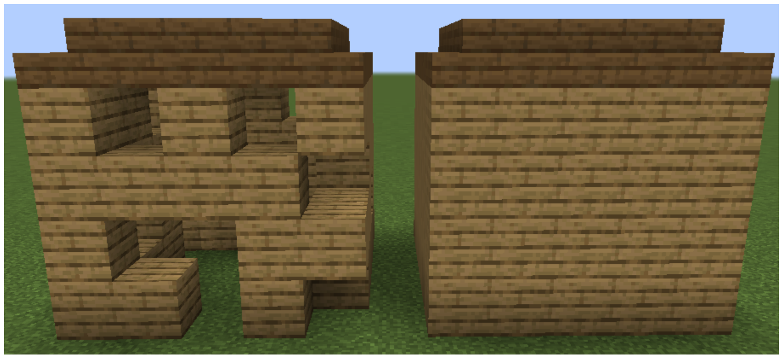


Figure 7 - Low (left) and high (right) bounding wall score structures.

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).

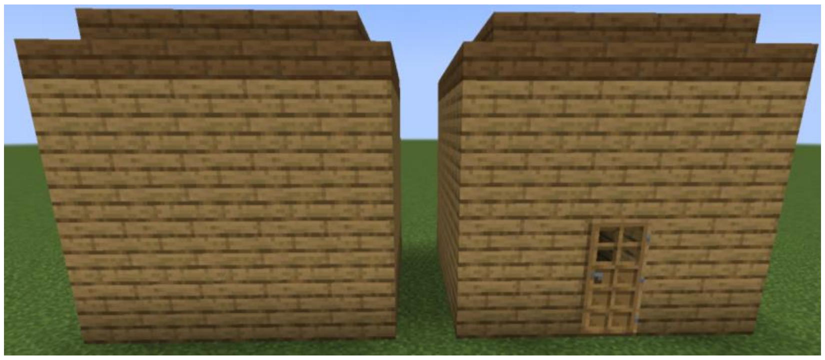


Figure 8 - Low (left) and high (right) door score structures.

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 prioritizes 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.

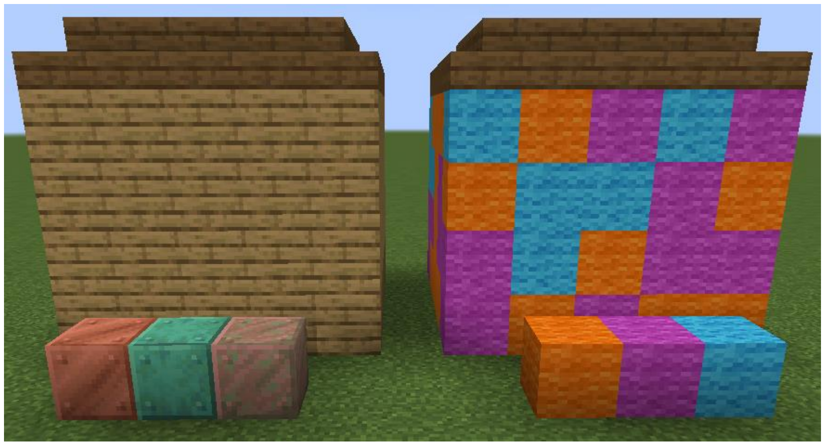


Figure 9 - Structures with their corresponding seed blocks in front. Low (left) and high (right) seed block score structures.

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 [18]. 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. Z

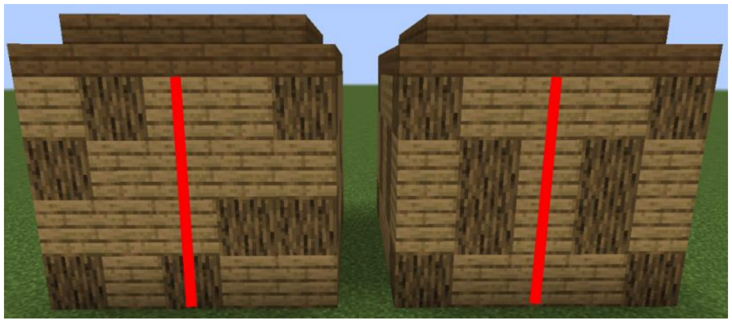


Figure 10 - Low (left) and high (right) symmetry score structures.

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 models using too many blocks and making a cluttered design, but not too few blocks.

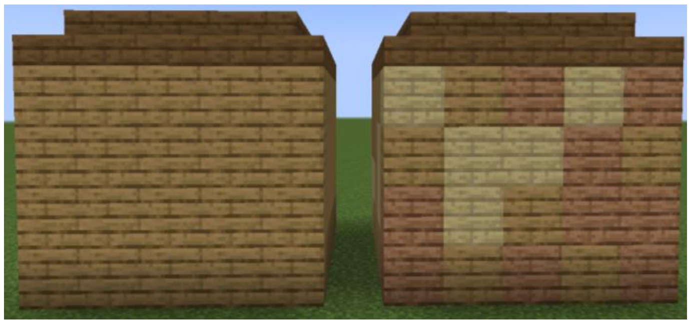


Figure 11 - Low (left) and high (right) block variance score structures.

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

3.4.1.2 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 subtle control. The structure scores:

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).

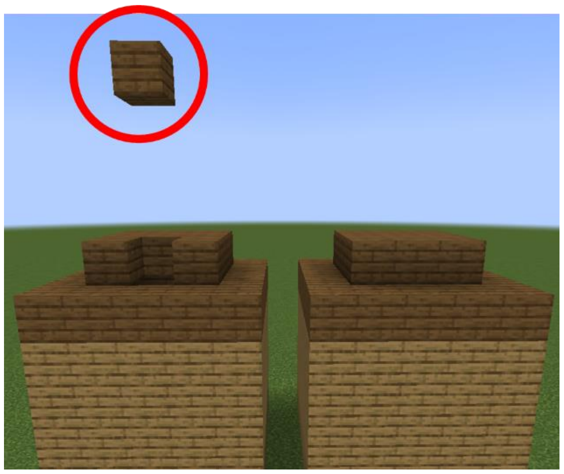


Figure 12 - Roofs with maximum height of 2 blocks. Left with compliance score of 0 (because of block circled in red). Right with a compliance score of 1.

Symmetry

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

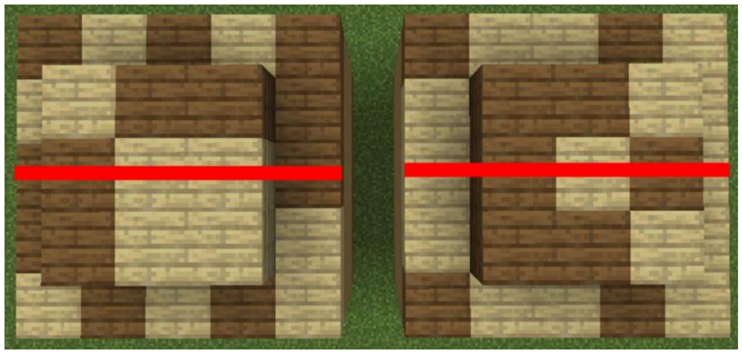


Figure 13 - Low (left) and high (right) symmetry score roofs.

Visual Complexity

This is the fitness for quality checking and making the solution more interesting. To make the solutions more interesting I am maximizing 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 14**Error! Reference source not found.**. I chose this solution over rewarding specific types of structure, like flat or triangle-sloped roofs, to allow for more flexibility.

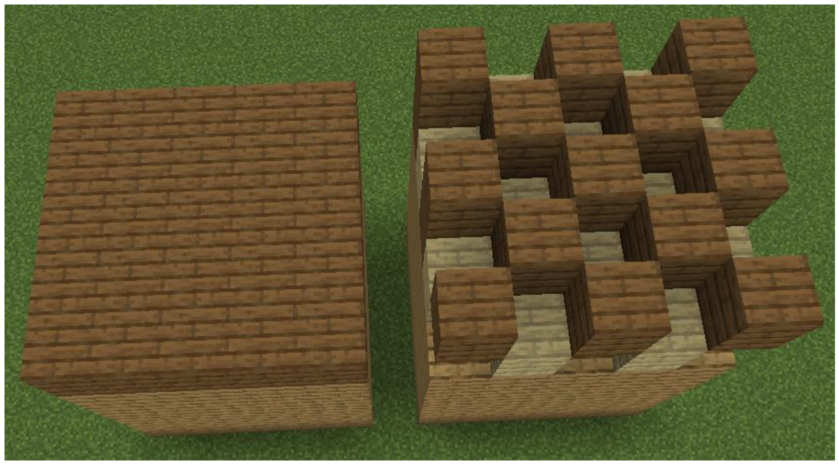


Figure 14 - Low (left) and high (right) visual complexity score roofs.

3.4.2 Novelty Scoring

To find the novelty of a genome, it needs to be compared to its closest relations. If a genome is very different to its closest relatives, then it will have a high novelty score. If an individual is compared to distant relatives as well, which are completely different, it will have an inflated novelty score. This means the novelty score will stop reflecting how the individual has changed, therefore an individual should 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 1. I chose a value of $k = \frac{2}{3} * size(population)$ which I felt would give the best comparison to the population.

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

Equation 1 - 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 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 earlier 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 [19]. 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.

3.5 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. What is important to note is that since the roof model only predicts block heights, not block types, when a building is generated the type of block to use is randomly selected from a subset of all blocks which I have curated. 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 changes 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 keeping the structural form, therefore this needs to be measured across the city. This is done with the following metrics.

Structure Score Distribution

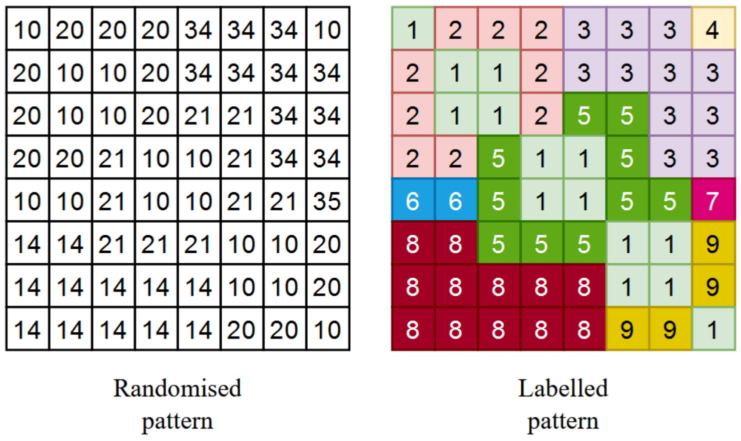
The structure score is my measurement for the structural form of a building. If all the generated structures in a city have a low structure score, then the constructions will be poorly made and will look worse.

Block Distribution

The visual quality of the structures is 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.

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 [20] (Figure 15). 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 Figure 15. By labelling the connected regions in the image it is much easier to identify patterns between structures, regardless of 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

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

Figure 15 - Randomized pattern being labelled using skimage labelling function.

4. Evaluation

4.1 Experimentation

The aim of the experimentation is to understand the effects of novelty on NEAT for building diverse building structures. To calculate an individual's fitness, the individual is used to create a structure, which is then scored using the structure score functions. The individual then has their novelty score calculated.

I will be running three experiments: no novelty, low novelty, and high novelty. To vary the level of novelty in each experiment, each one will have a different novelty:structure score weighting. The weightings for each experiment are shown in Figure 16. For each experiment two runs of training will be done. This is necessary since the model evolution is inherently random and can vary significantly.

Experiment	Structure Score Weight	Novelty Score Weight
No Novelty	1	0
Low Novelty	0.8	0.2
High Novelty	0.2	0.8

Figure 16 - Table showing structure and novelty score weights for each experiment

Each house experiment had a population of 20 and each roof experiment has 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 experiment was trained until the max structure score hadn't improved over the last 50 generations but was given a grace period of 100 generations where it wouldn't be stopped. This was necessary because, especially early in the training, there are many outliers which regularly stopped the training at 50 generations.

4.2 Training Comparisons

A summary of the training is shown in Figure 17 - Figure 20, and full training graphs are shown in Figure 32 - Figure 37. Figure 17 & Figure 18 show the maximum population average for each experiment. We are interested in the population average because when it comes to generating structures, the whole population will be used. It is important to use the whole population to get the diversity, improved through novelty search, to come through in the structures. Therefore, the population must be viewed as a whole. Figure 19 & Figure 20 shows the number of generations taken for each experiment to reach a milestone structure score (0.4, 0.5, …, 0.8).

The maximum population average scores (Figures 17 & 18) shows that the high novelty experiment outcompetes the other experiments in both model while the no novelty and low novelty experiments show roughly the same results in both models. The exact values of these structure scores should be taken with large uncertainty as each individual is only tested on one structure, which is a low sample size and prone to error. When the experiments are tested on many structures the values will be much more accurate.

Combing this information with Figures 19 & 20 gives more evidence towards the high novelty experiments out competing the other experiments. The high novelty experiment regularly reaches a higher structure score and tends to do this in a significantly faster time. What is interesting is that, in Figure 19, while both runs of the high novelty model reach roughly the same result, run 2 takes ~20x longer. Along with the run 1 of the low novelty experiment, which doesn't reach close to run 2, we can see that the performance of the algorithm is significantly random and changes drastically between runs.

House Model	No Novelty		Low Novelty		High Novelty	
Run	1	2	1	2	1	2
Max pop. avg.	0.681	0.588	0.427	0.619	0.733	0.772

Figure 17 - Table showing maximum population average reached by each novelty experiment in the house model.

Roof Model	No Novelty		Low Novelty		High Novelty	
Run	1	2	1	2	1	2
Max pop. avg.	0.739	0.659	0.704	0.696	0.854	0.817

Figure 18 - Table showing maximum population average reached by each novelty experiment in the roof model.

	House Model	No Novelty		Low Novelty		High Novelty	
		Run 1	Run 2	Run 1	Run 2	Run 1	Run 2
Structure Score	0.4	0	4	1	11	0	0
	0.5	3	7	-	14	0	0
	0.6	17	-	-	107	20	371
	0.7	-	-	-	-	29	564

Figure 19 - Table showing the number of generations taken for each experiment's population average to reach a milestone structure score (house model).

	Roof Model	No Novelty		Low Novelty		High Novelty	
		Run 1	Run 2	Run 1	Run 2	Run 1	Run 2
Structure Score	0.4	2	1	1	1	0	0
	0.5	33	22	22	40	1	8
	0.6	46	57	46	58	29	39
	0.7	109	-	128	-	36	64
	0.8	-	-	-	-	99	133

Figure 20 - Table showing the number of generations taken for each experiment's population average to reach a milestone structure score (roof model)

4.3 Multi-structure Comparisons

To populate the datasets for comparison, 20,000 structures are generated using each novelty experiment. Since some of the metrics rely on grouping building sizes, therefore it is important there is a large number of structures to properly fill all grouped datasets.

4.3.1 Structure Score Distribution

The structure score distributions are shown in Figure 21 & Figure 22. Comparing between the graphs, we can see that the house model scores are much lower than the roof model scores. This could be because the search space for the house model was much more complex or that the population of the house model was significantly smaller, reducing the chances for positive evolutions.

As expected, in both models, the no novelty experiment has performed worst, and the high novelty model has performed best. Interestingly, in most of the experiments, there are many outliers spread across a wide range of scores, showing the model has still not perfected the constructions. However, the high novelty experiment (and low novelty to an extent) are much more top weighted, showing a more consistent performance.

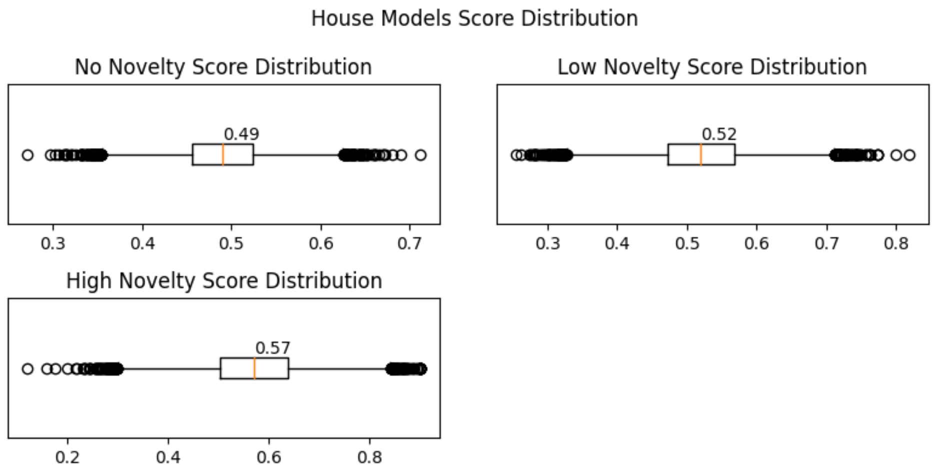


Figure 21 - Score distribution of generated houses for different models

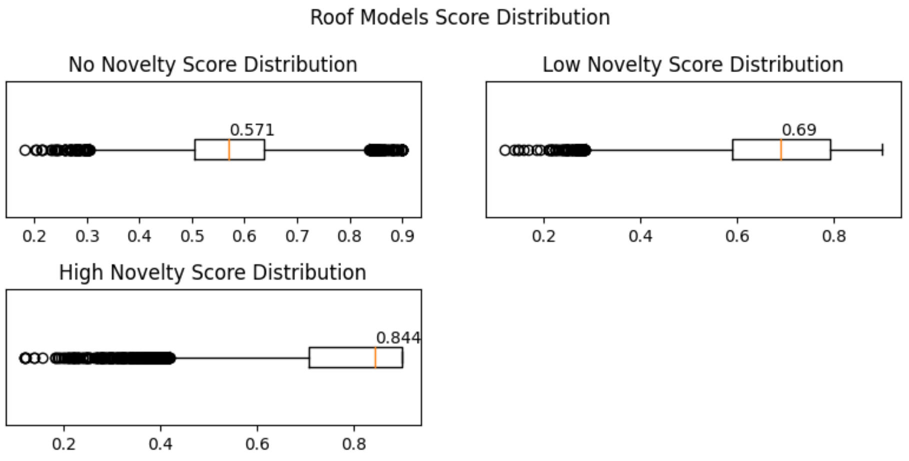


Figure 22 - Score distribution of generated roofs for different models

4.3.2 Block Distribution

Block distribution describes the distribution of blocks used in the construction of houses. From Figure 23, we can see the higher the novelty, the higher the spread of blocks used. This could be because the novelty search has encouraged a wider range of blocks to be used. Without novelty search, the model has settled on a much smaller subset of blocks which it could use to reach an adequate score. While the high and low novelty models have used a much wider range of blocks, it is still a small subset of potential blocks it could use. The high novelty experiment used $\frac{31}{132}$ blocks, which means that a higher level of novelty might still improve this further.

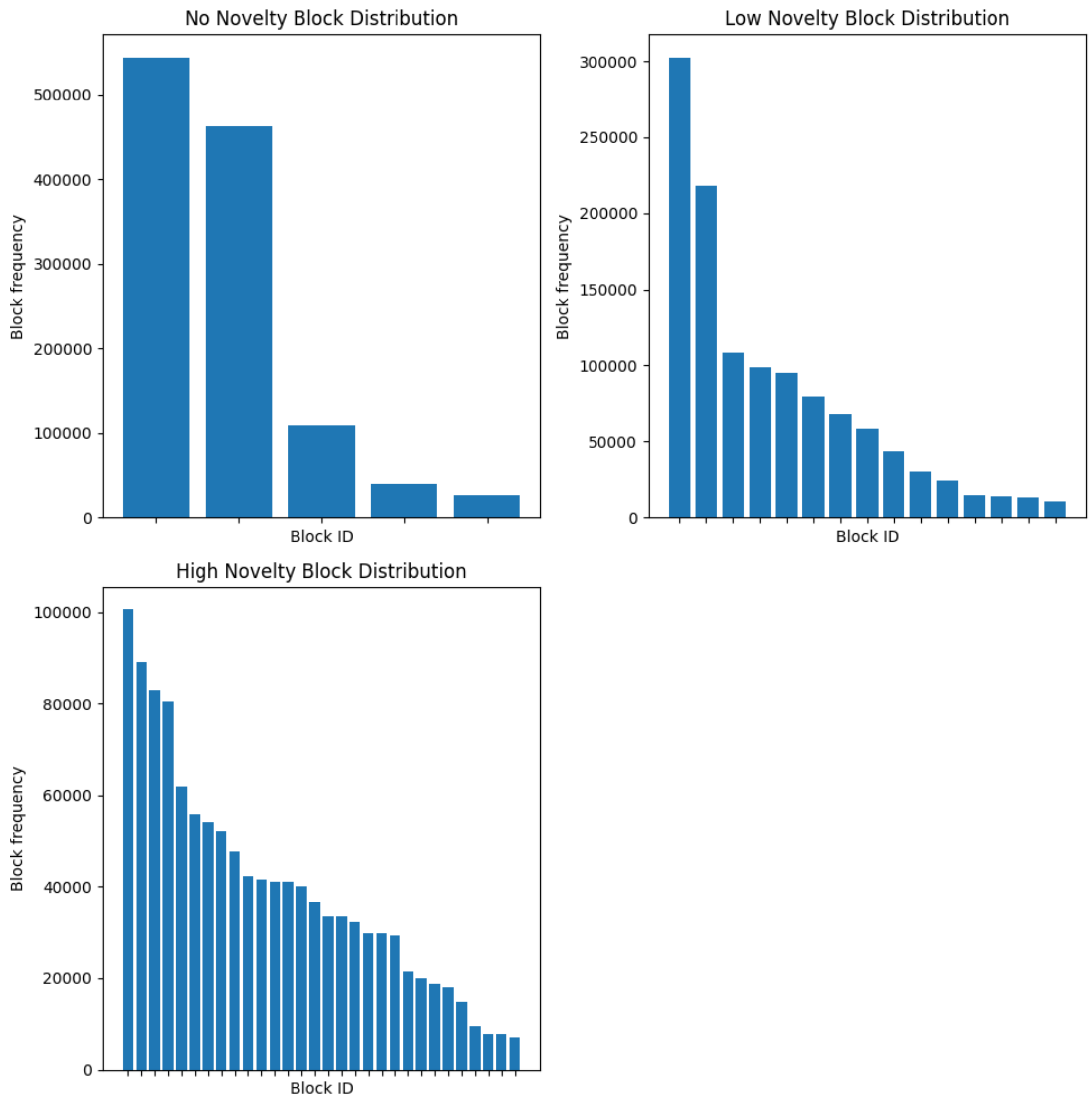


Figure 23 - Block distribution of generated structures for different models

4.3.3 Pattern Variety

The pattern variety gets the percentage pattern difference between structure patterns, or $1 - pattern\_similarity$. Therefore, 1 is complete variation between all structures and 0 is completely the same. The patterns are independent of the blocks used. Again, as expected, the high novelty experiment has outperformed the other experiments in both models. But surprisingly the low novelty model has performed much better than expected. Given a higher than expected variance score but lower structure score this indicates that the model is placing blocks more randomly, without much structural form. This will show in the visual analysis. It is hard to say whether the models would benefit from a more novelty, and therefore more variance. Pattern variance is a tradeoff, too much and the buildings will look random and formless, too little and it loses its diversity. As Koster[21], explains, there is a "golden zone" between novelty and familiarity which is most pleasing to people. Since this is mostly subjective, refining the best variance score would require humans to decide what is most pleasing. Without that, maximizing for variance and structure score will have to suffice.

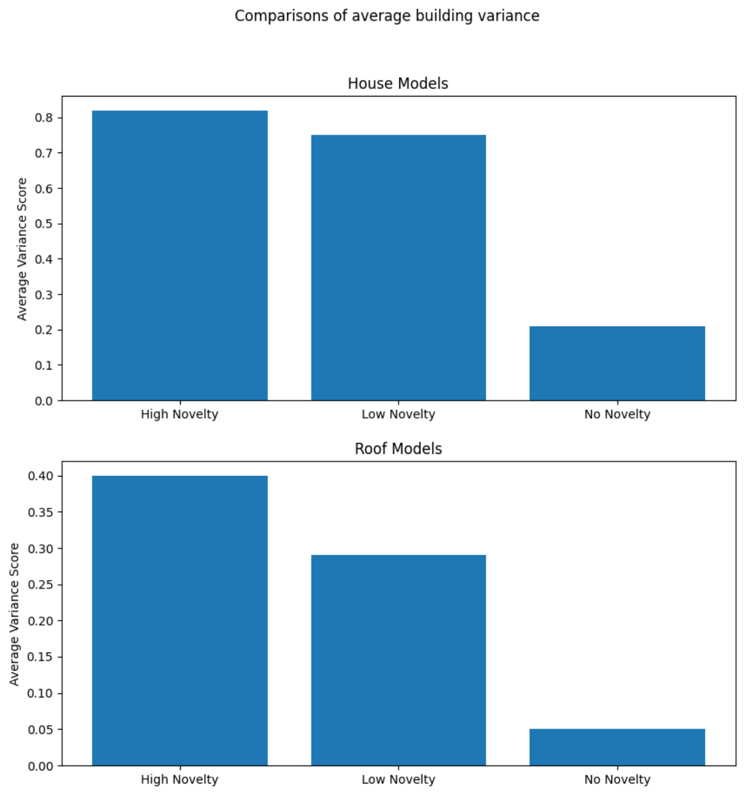


Figure 24 - Variance in building patterns for different models

4.4 Visual Analysis

A sample of interesting houses are shown in Figure 25, a larger sample is shown in Figure 388. From the larger sample we can see there is very little variation. Many houses have flat roofs, with very low visual complexity. A vast majority of the buildings are composed of one type of block. I theorize this is because this reaches a relatively good structure score. If a building is made of one block type, then it must always be perfectly symmetrical. If another block is introduced, it will probably have a low symmetry. Shown in Figure 25 are some of the houses which have more than one block. They have almost no symmetry and look randomly placed. The roof model also hasn't learnt not to place blocks randomly in the air.

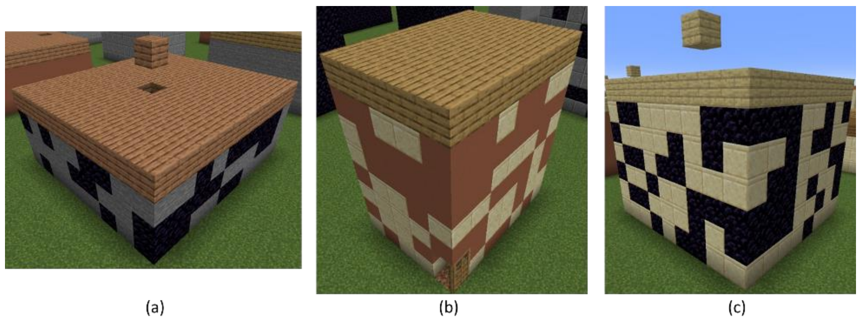


Figure 25 – Sample of houses taken from no novelty generated structures.

A sample of interesting houses and roods are shown in Figure 26 & Figure 27, a larger sample is shown in Figure 399. Figure 39 shows a big improvement by the low novelty model. Most buildings use more than one block and there are some doors being placed on the ground. The roofs also have much more visual complexity and are more diverse. The houses and roofs are still lacking much symmetry, and as previously mentioned I believe this is because the model is placing blocks more randomly, without much knowledge of symmetry. While this is the case over many structures, Figure 27 shows a couple of examples where the symmetry is good in some circumstances, so there is some knowledge of symmetry developing. Figure 26a shows a house which has a very low bounding wall fitness, which I considered a problem the models has solved, but there are still some instances where it fails to accomplish this.

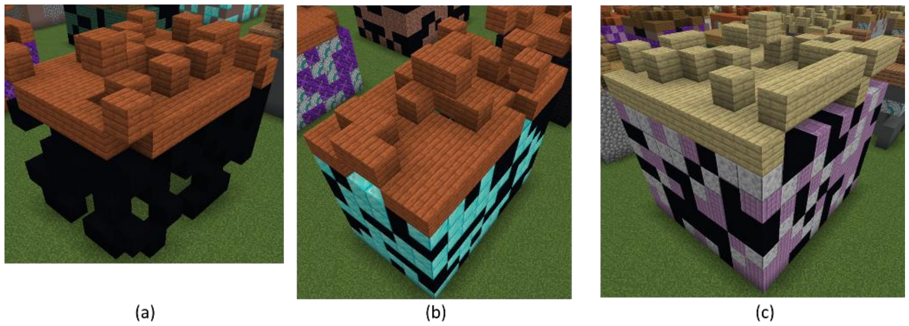


Figure 26 - Sample houses taken from low novelty generated structures

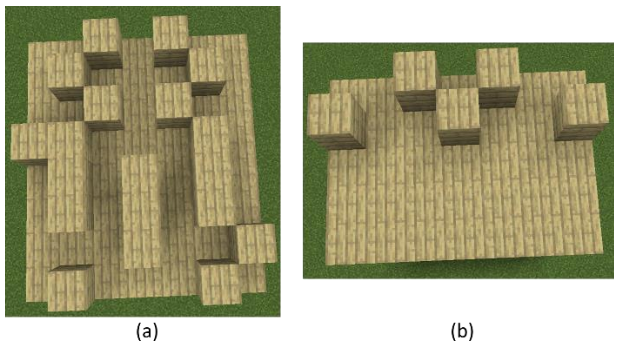


Figure 27 - Sample roofs taken from low novelty generated structures

A sample of interesting houses and roods are shown in Figure 28 & Figure 29, a larger sample is shown in Figure 3940. This again shows a significant improvement over the previous generation. There is a significant amount more symmetry, especially in the roof model. Most of the roofs are almost perfectly symmetrical and show much more visual complexity. I think this most has almost perfected the visual complexity metric, as I don’t believe it is possible to always achieve perfect visual complexity. Some examples of houses are shown in Figure 28, where the symmetry has is perfect, but this is not the case for all the houses, so there is some room for improvement there. Figure 28a also shows a bad bounding wall score. This is much less common in these structures but shows it can still happen.

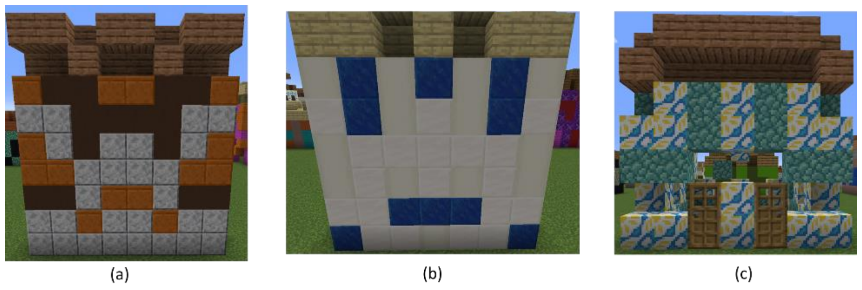


Figure 28 - - Sample houses taken from high novelty generated structures.

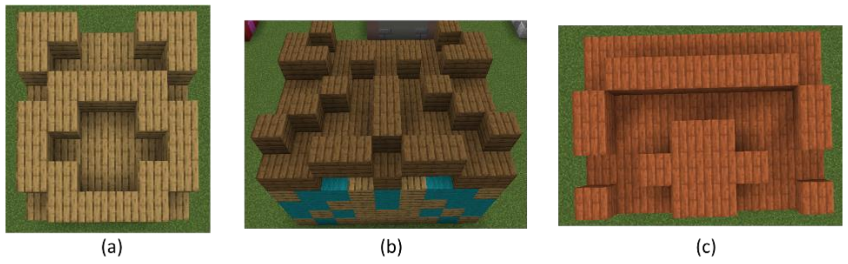


Figure 29- Sample roofs taken from high novelty generated structures.

5. Overview and Future Work

Overall, I think this project has been successful. The experimentation has clearly shown that using novelty search has increased the diversity in the models, improving the results. There are plenty of avenues to explore off the back of this project. It has been shown that the correct balance between novelty and performance score (structure score in this context) is important. To improve upon this there needs to be more definition in this by experimenting with more levels of novelty. Another improvement would be to do more runs of each model. In most models there are significant differences between the runs, and increasing the number of runs would give more confidence in the results. In retrospect, the value I chose for KNN ($\frac{2}{3}$ of the population) was too high and would probably capture too much of the population, giving an inaccuracy novelty value. Using M. Beukman *et al.* [13] as a guide, a value of $K = \frac{1}{4}$ would have been must more suitable.

In terms of the structure output, improvements to the complexity of the structures would be a good visual improvement. Adding other elements, like windows, chimneys, gardens, roads, would go a long way to improving the complexity and diversity of the city. Another important aspect to improve would be the color. Currently the colors are randomly placed together, leading to some poor looking results. Leveraging some color theory and giving the model block color information would go a long way to improving the look of the constructions.

Personally, my knowledge about machine learning has significantly expanded. Before this dissertation I had little knowledge of genetic algorithms and unsupervised deep learning techniques. I also feel my Python skills have improved. While I have been using Python for many years, I have not done many big object-oriented projects like this before, which has been enlightening.

References

[1] A. Newhouse, "Making sense of no man's sky's massive universe." GameSpot, Aug. 2016. [Online]. Available: https://www.gamespot.com/articles/making-sense-of-no-mans-skys-massive-universe/1100-6441344/

[2] Bohemia Interactive, "ARMA 3." Bohemia Interactive, Sep. 12, 2013. Accessed: May 18, 2023. [Online]. Available: https://arma3.com/

[3] P. Carlson, "Arma 3 map might be bigger than you ever imagined," *pcgamer*. PC Gamer, Aug. 2013. [Online]. Available: https://www.pcgamer.com/arma-3-map-might-be-bigger-than-you-ever-imagined/

[4] A. Summerville *et al.*, "Procedural Content Generation via Machine Learning (PCGML)." 2018.

[5] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evol Comput*, vol. 10, no. 2, pp. 99–127, 2002, doi: 10.1162/106365602320169811.

[6] J. Lehman and K. O. Stanley, "Novelty Search and the Problem with Objectives," 2011, pp. 37–56. doi: 10.1007/978-1-4614-1770-5_3.

[7] M. Persson, "Welcome to the minecraft official site," *Minecraft.net*. Mojang Studios, Apr. 2023. [Online]. Available: https://www.minecraft.net/en-us

[8] D. Grbic, R. B. Palm, E. Najarro, C. Glanois, and S. Risi, "EvoCraft: A New Challenge for Open-Endedness." 2020.

[9] S. Whitworth, "How big is a minecraft world?," *Sports news*. Sportskeeda, Oct. 2021. [Online]. Available: https://www.sportskeeda.com/minecraft/how-big-minecraft-world

[10] Z. Wei and Ohanlon Martin, "RaspberryJuice." SpigotMC, May 2016. Accessed: May 18, 2023. [Online]. Available: https://www.spigotmc.org/resources/raspberryjuice.22724/

[11] M. Ohanlon, "McPi." May 2018. Accessed: May 18, 2023. [Online]. Available: https://github.com/martinohanlon/mcpi

[12] D. Grbic, R. B. Palm, E. Najarro, C. Glanois, and S. Risi, "Simple Minecraft Evolver." Feb. 2022. doi: 10.1007/978-3-030-72699-7_21.

[13] M. Beukman *et al.*, “Hierarchically Composing Level Generators for the Creation of Complex Structures.” 2023.

[14] A. Joy, “Pros and cons of reinforcement learning,” *Pythonista Planet*. Mar. 2019. [Online]. Available: https://pythonistaplanet.com/pros-and-cons-of-reinforcement-learning/

[15] M. Park, M. Lee, and S. Yu, “HRGAN: A Generative Adversarial Network Producing Higher-Resolution Images than Training Sets,” *Sensors*, vol. 22, no. 4, p. 1435, Feb. 2022, doi: 10.3390/s22041435.

[16] J. Hui, “Why it is so hard to train generative adversarial networks!” Medium, Jun. 2018. [Online]. Available: https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

[17] V. Mallawaarachchi, “How to define a fitness function in a genetic algorithm?,” *Towards Data Science*. Medium, Nov. 2017. [Online]. Available: https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4

[18] Y. Huang, X. Xue, E. Spelke, L. Huang, W. Zheng, and K. Peng, “The aesthetic preference for symmetry dissociates from early-emerging attention to symmetry,” *Sci Rep*, vol. 8, no. 1, Apr. 2018, doi: 10.1038/s41598-018-24558-x.

[19] A. Salehi, A. Coninx, and S. Doncieux, “Geodesics, Non-linearities and the Archive of Novelty Search.” 2022.

[20] S. van der Walt *et al.*, “scikit-image: image processing in Python,” *PeerJ*, vol. 2, p. e453, May 2014, doi: 10.7717/peerj.453.

[21] R. Koster, *A theory of fun for game design*, Second Edition. 2004.

s

Appendix

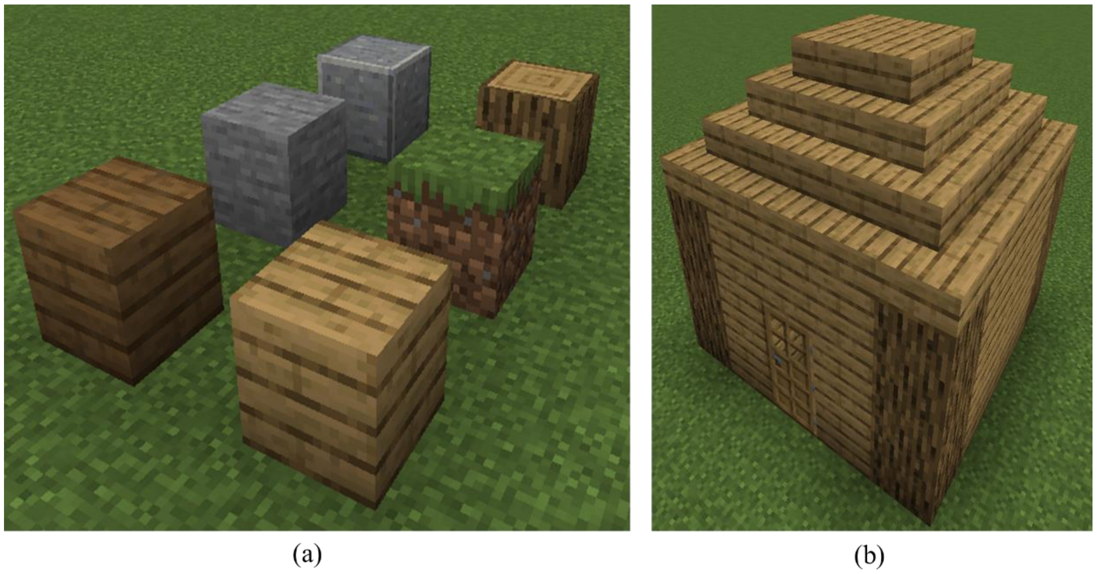


Figure 30 - (a) Sample of blocks (voxels) from Minecraft [7] *(b) Example structure built from blocks.*

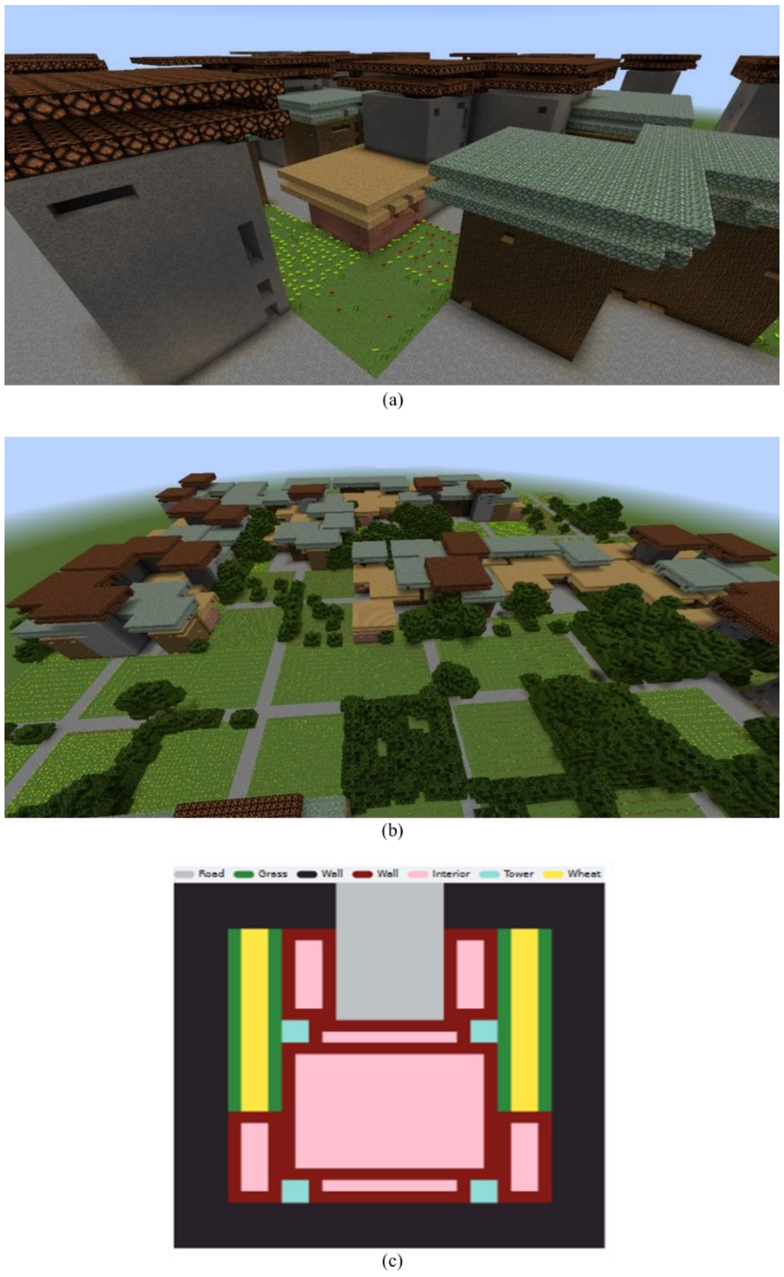


Figure 31- Examples from EvoCraft PGCNN [13]. *(a) Generated house, roof, garden inside a town (b) Birdseye view of a procedurally generated town (c) 2D tilemap plan for a town*

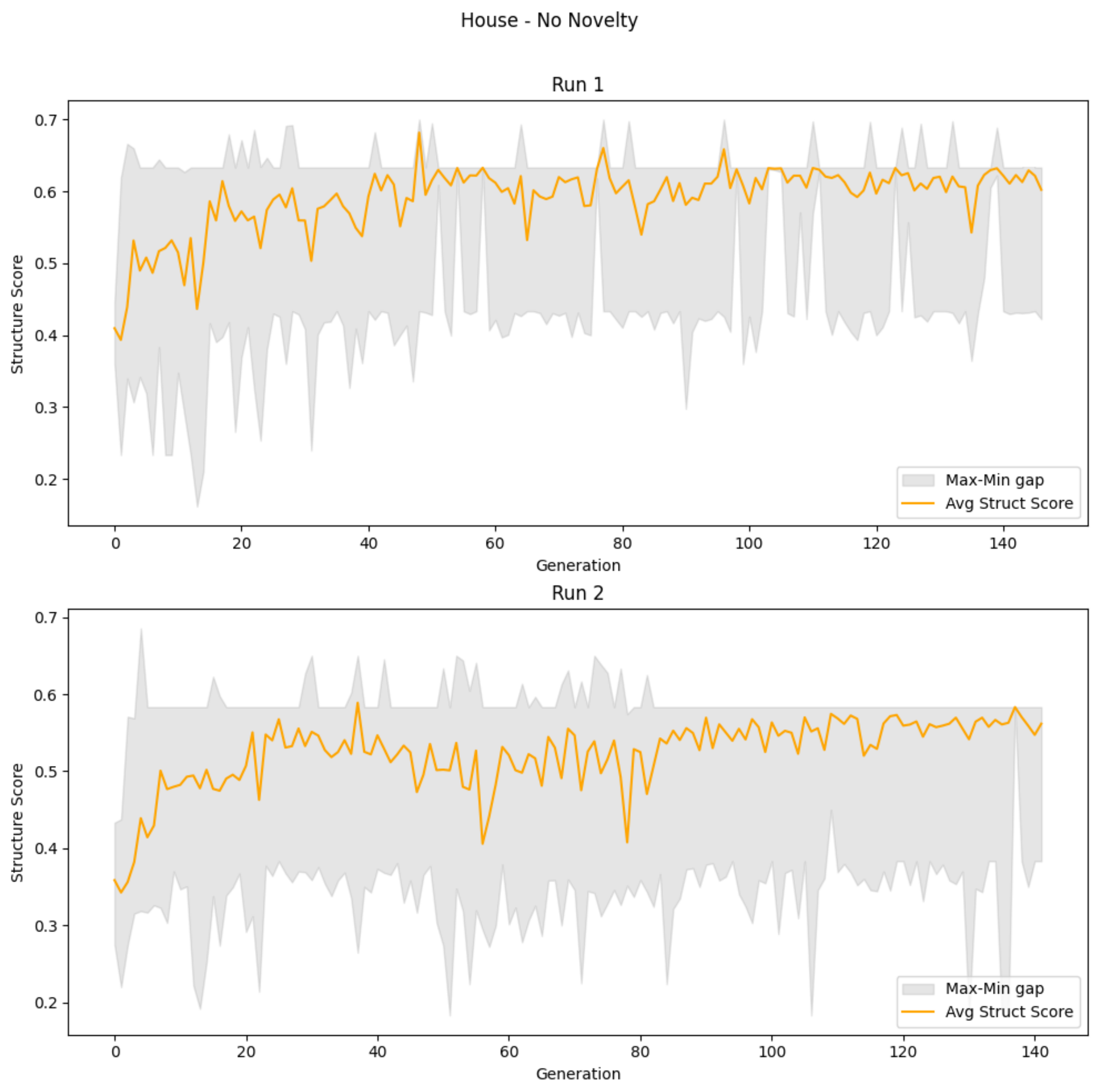


Figure 32 - No novelty experiment showing structure score values during training (house model).

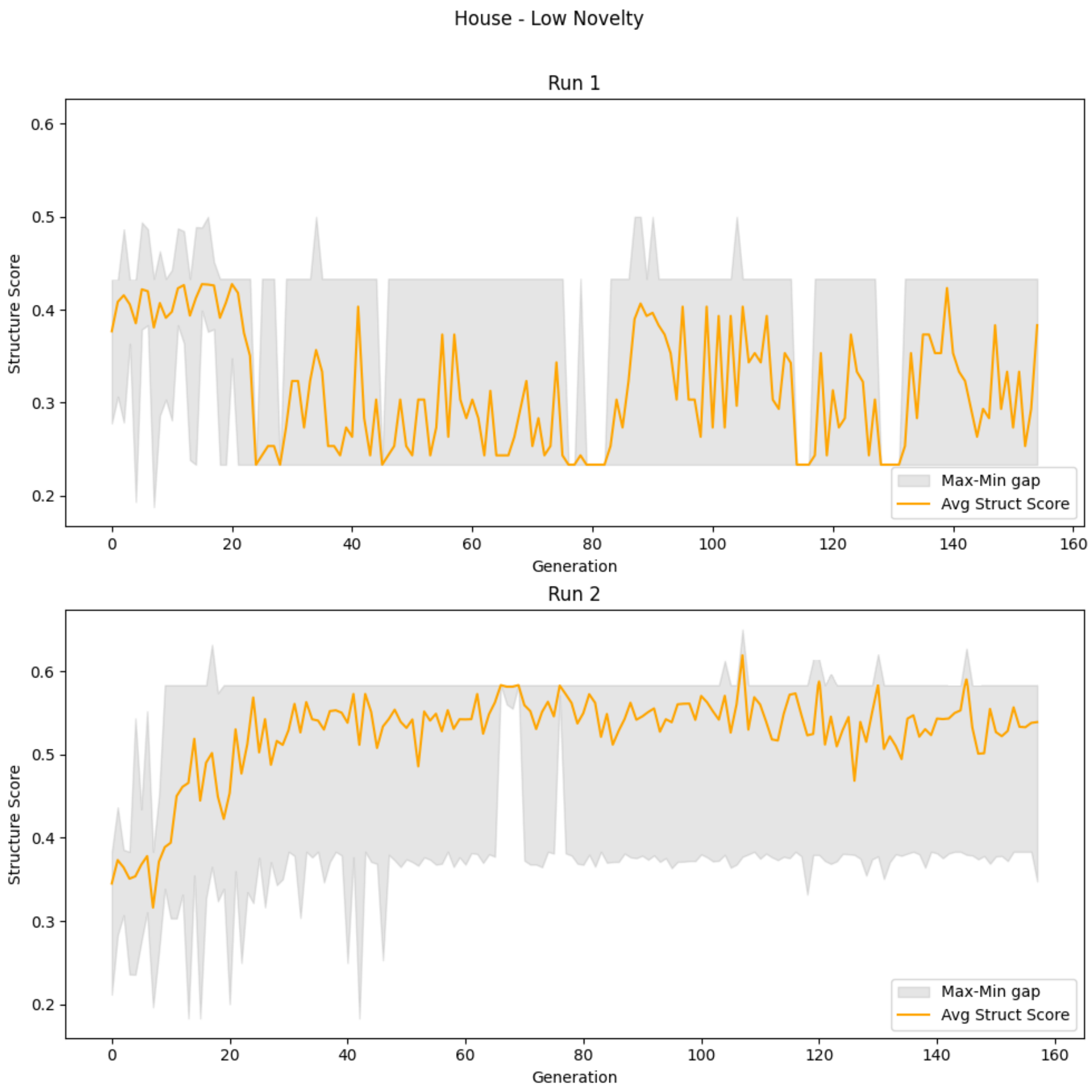


Figure 33 - Low novelty experiment showing structure score values during training (house model).

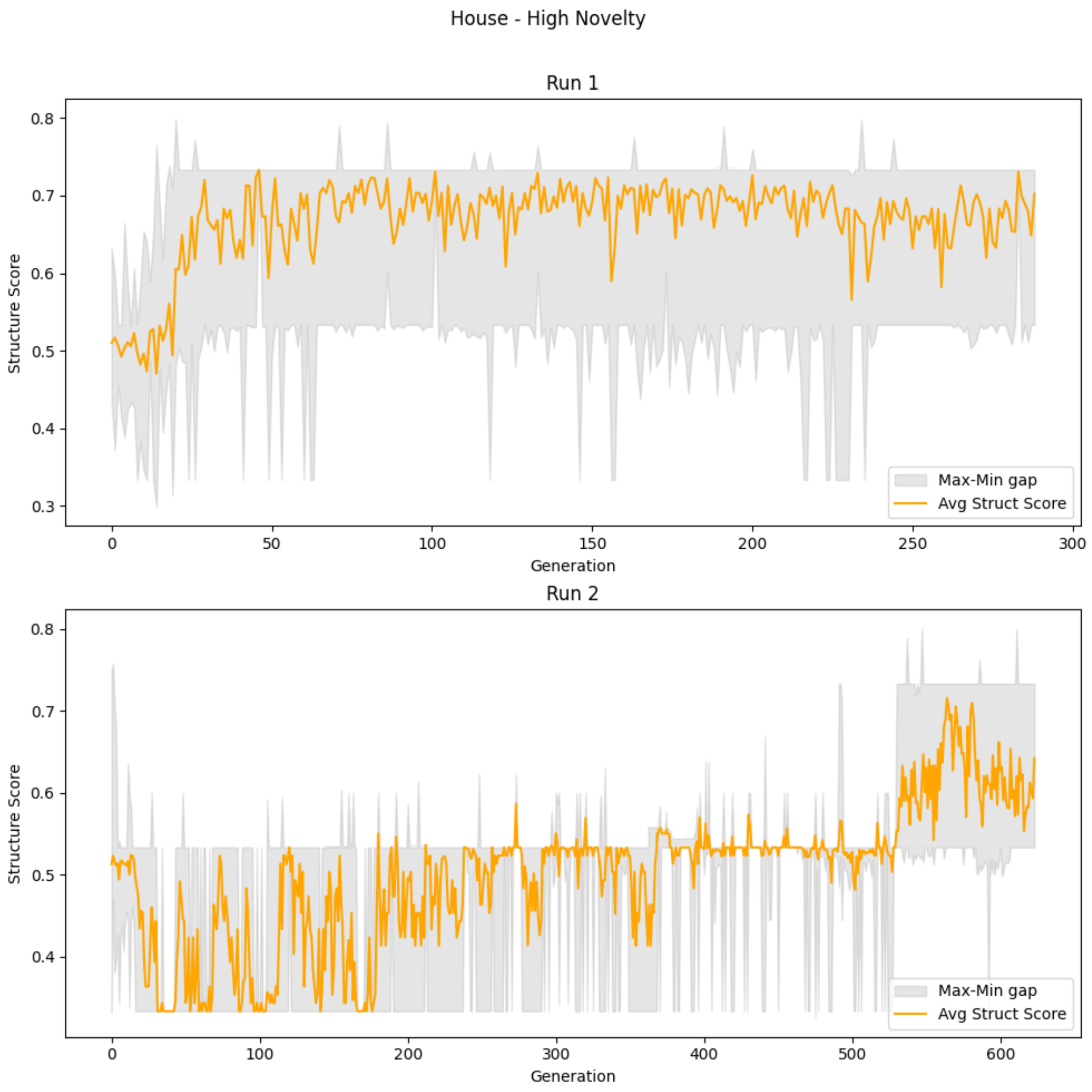


Figure 34 - High novelty experiment showing structure score values during training (house model).

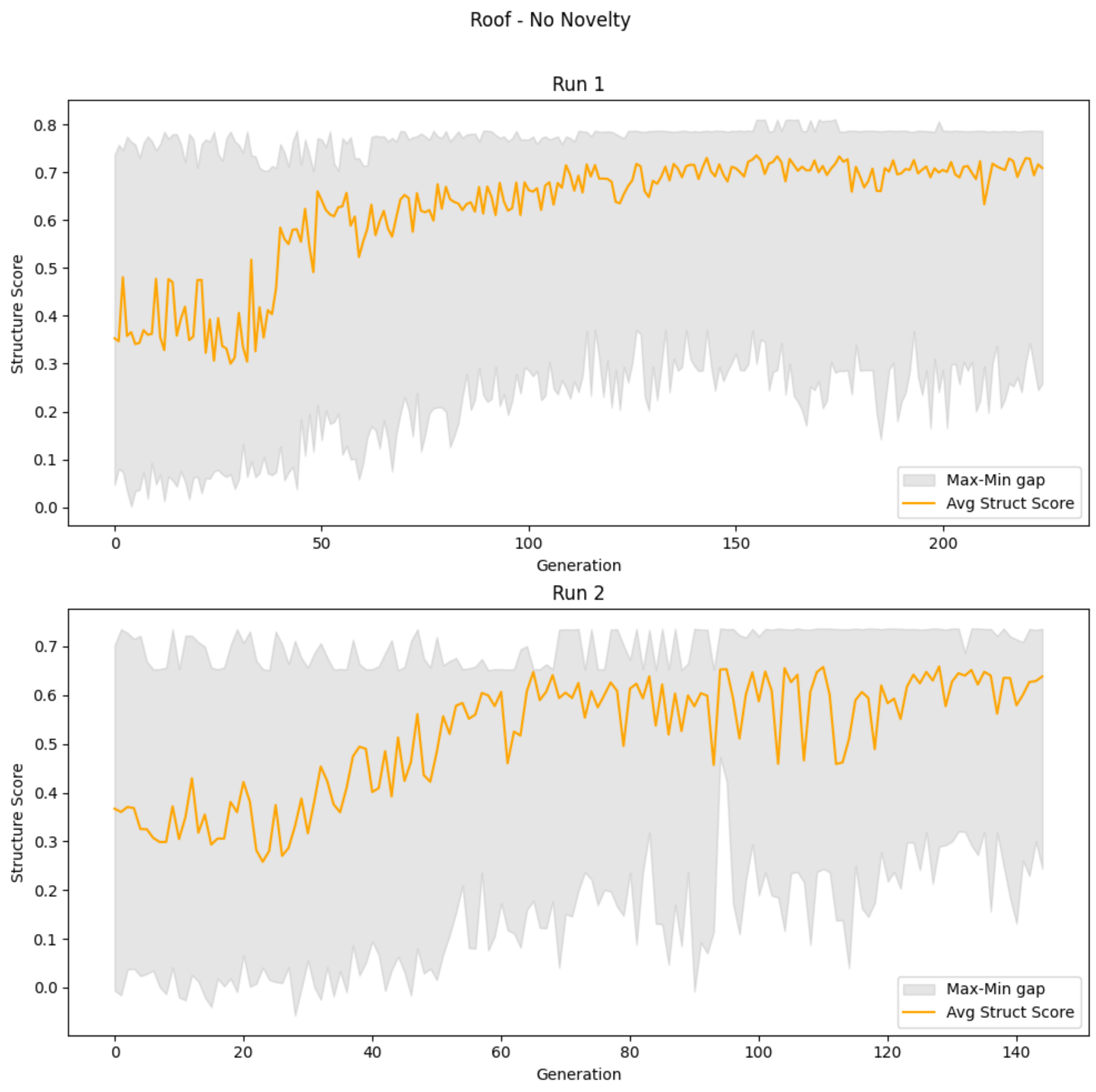


Figure 35 - No novelty experiment showing structure score values during training (roof model).

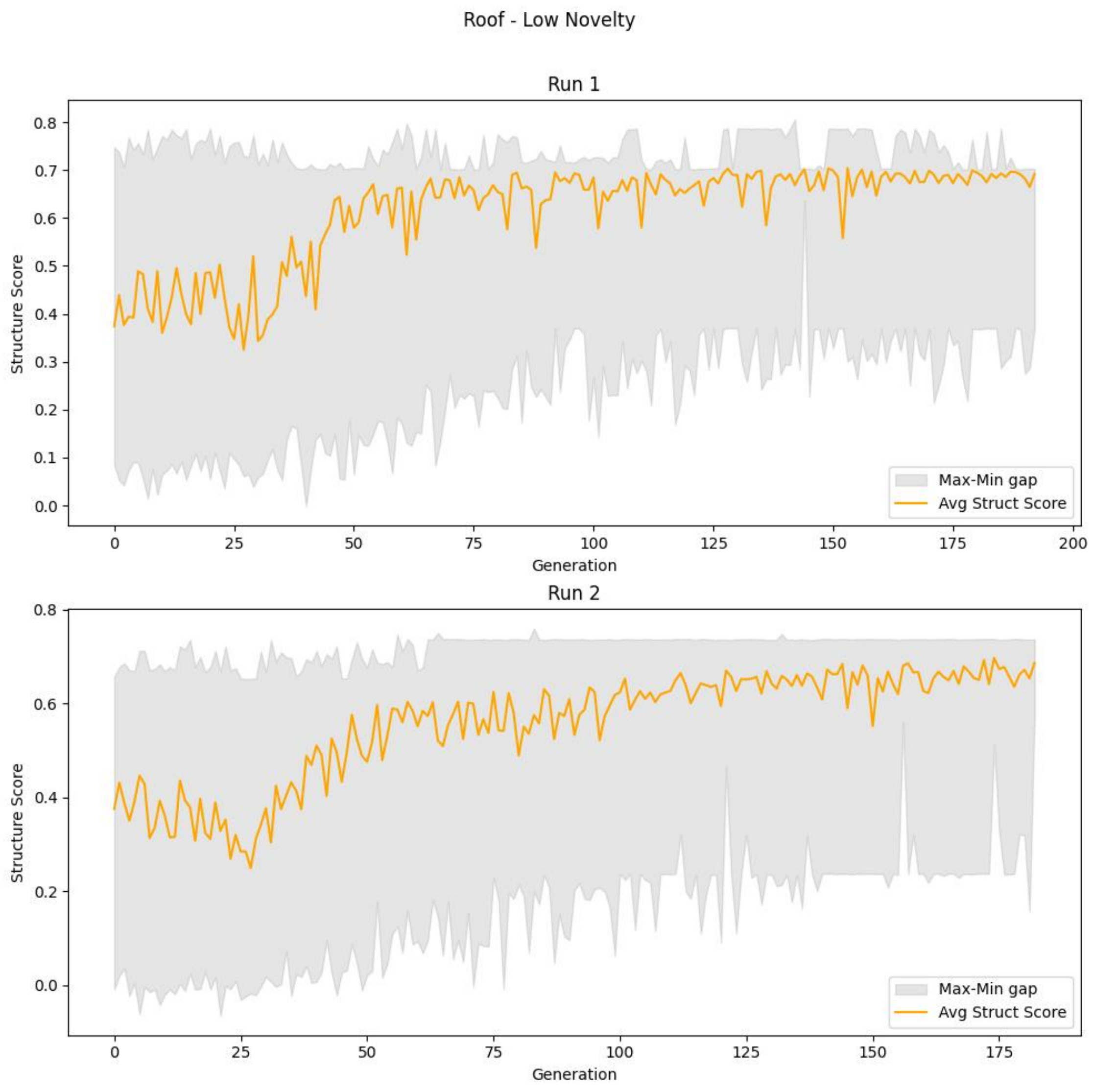


Figure 36 - Low novelty experiment showing structure score values during training (roof model).

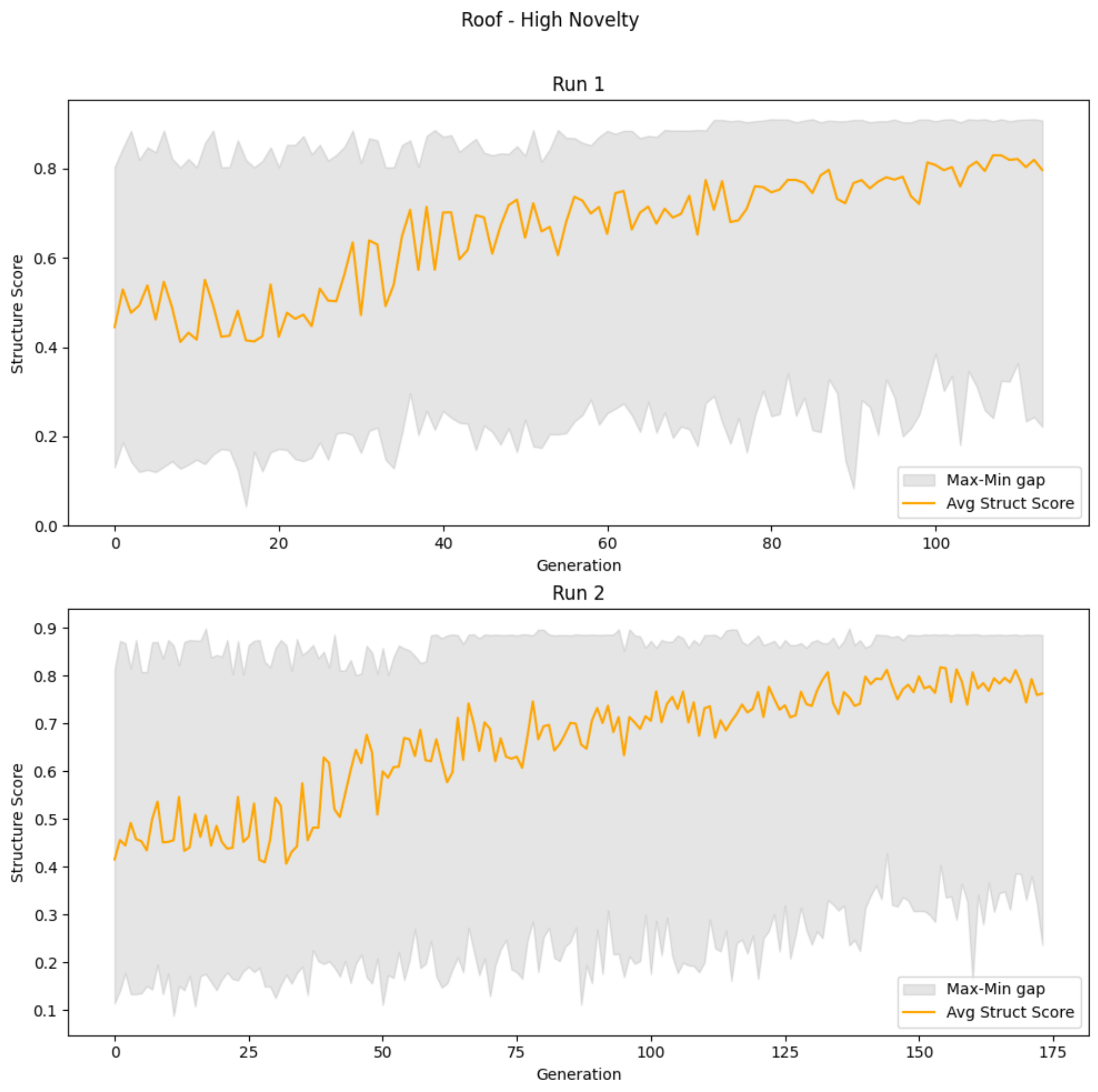


Figure 37 - High novelty experiment showing structure score values during training (roof model).

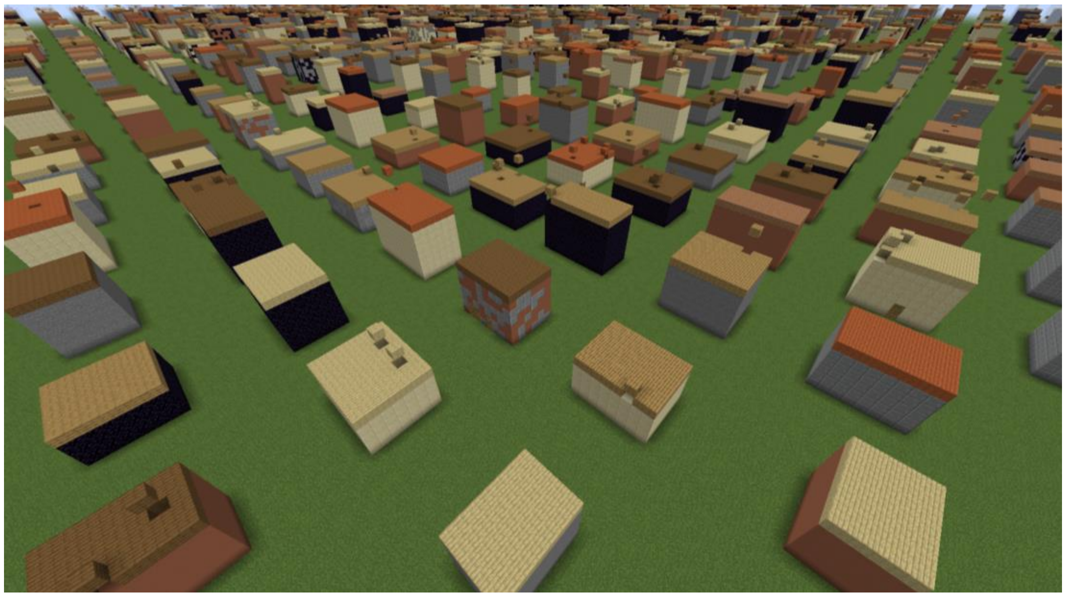


Figure 38 - Birdseye view of structures generated by no novelty models

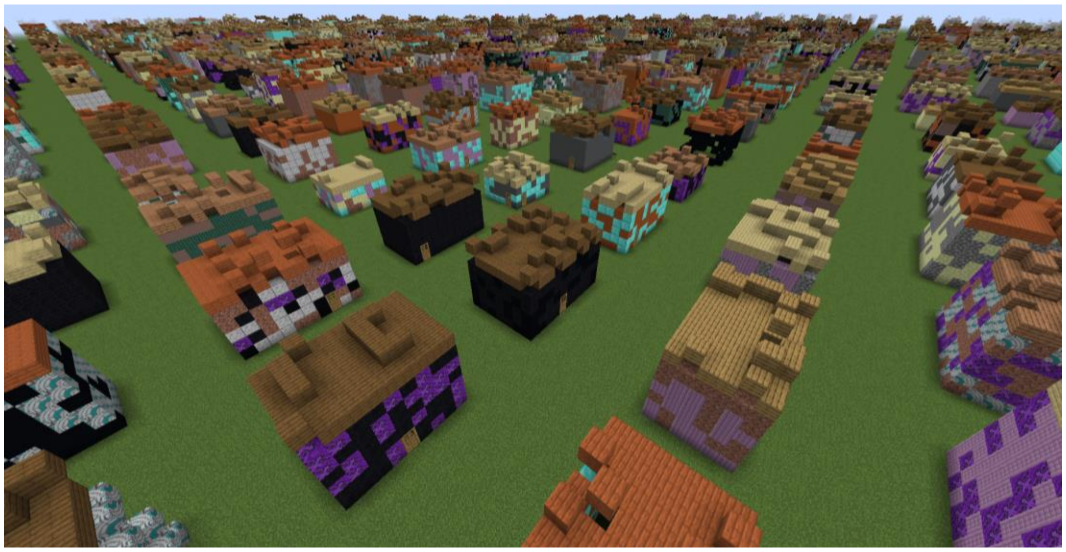


Figure 39 - Birdseye view of structures generated by low novelty models

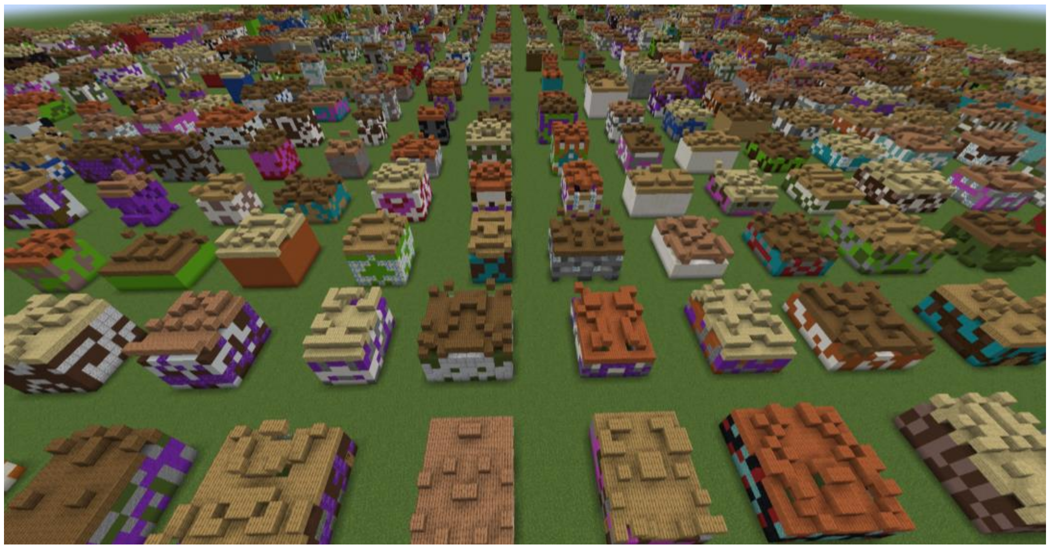


Figure 40 - Birdseye view of structures generated by high novelty models