Generation of Stable Structures using Neural Cellular Automata

Lucas Petersen, Mikkel Ditlevsen, Oliver Astrup
lupe@itu.dk, midi@itu.dk, olas@itu.dk
Modern Artificial Intelligence
KGMOARI1KU

Abstract

Neural cellular automata has previously been applied most often to a 2D domain. Recent research has shown that they can also be effective if extended to a 3D environment, and have been capable of generating structures in such a space. This paper presents a use case for generating 3D structures which have some particular properties. Specifically, being stable under the force of gravity. This is done by using neural cellular automatas, trained via an evolutionary search algorithm, to generate 3D structures. These structures are evaluated based on a fitness function that utilises Unity's physics engine to simulate the force of gravity on the structure. This is an early stage of experimentation, but the end goal is to have NCAs that are capable of generating stable structures that could be useful for procedural content generation in games and other virtual 3D environments. This paper shows that NCAs are capable of generating these structures, though more work needs to be done before these NCAs can be used for procedural content generation.

Introduction

NCAs (Neural Cellular Automata) are a specific kind of CAs. Traditionally CAs has been used in a 2D environment, where a small set of simple rules, governs the evolution of cells in a grid. Probably most famously to the general public as Conway's Game Of Life (Gardener, 1970). NCAs differ, because the rules are no longer specified, but instead a neural network governs the rules. The network takes as input a grid of cells, and output the sames grid of cells with an update applied by the network. NCAs can in principle be used to generate anything that can be represented in a grid-like structure, and is not contained to only 2 dimensions.

This paper will present an attempt to use NCAs to generate 3D structures that are stable under the force of gravity. The structures will be a set of vertices represented as solid spheres in a 3D space, connected by a set of edges represented as solid cylinders. For simplicity the structure will only be able to break under gravity at the connection between the cylinders and the vertices.

Though this paper presents the early stages of experimentation, this could be relevant to for example procedurally generating structures in games and other virtual 3D environments.

Background

The generation of structures using an NCA has been shown to work previously by Sudhakaran et al. (2021), where an NCA has been trained on the reconstruction of both simple and complex structures within the game of Minecraft. This project made use of the 3 dimensional grid of Minecraft to represent the entities that it was trying to reconstruct, and represented each cell in the grid separately. In that case and also typically, the NCAs have been trained by supervised learning using gradient descent. Another good example of using gradient decent is by Mordvintsev et al. (2020), in which they use a convolutional neural network combined with different per cell operations to reach their goal of generating self-repairing 2D images. This is however not feasible for this project, as a large dataset of the structures this project should generate does not exists, and therefore loss functions aren't an option.

Using unsupervised learning to train NCAs has been shown to work by Earle et al. (2021), in which they used an evolutionary quality diversity algorithm to train an ensemble of NCAs for generating 2D levels for games. This project was inspired by their approach, although quality diversity isn't used in this project

The idea of using evolutionary search algorithms to train neural networks (neuroevolution) is not new, and there exists a range of different approaches and methods. A collection of some of these can be found in a paper by Floreano and Mattiussi (2008), which summarises and categorises them. Although neuroevolution can be used both for evolving the topology of the network and the hyperparameters, this paper focuses on the latter.

During the research for this paper no previous papers were found that uses neuroevolution to evolve NCAs capable of generating 3D structures which are stable under gravity.

Implementation

This section will describe the implementation of the project and the methods that were used. It will also go into detail about some of the previous implementation attempts and explain why they didn't work or why it was decided to change them.

The Physics Environment

To test the stability of the structure, simulations of the force of gravity being applied to the structure was used. The stability results are then used for evaluating each NCA. To run the simulation the Unity game engine (Unity Technologies, 2005) was chosen, even though it contained many elements that would not be needed for this implementation. It did contain its own physics engine that was easy to access and can be turned on and off when it's needed, so that physics would only be applied when the structures were evaluated. Thereby the structures did not have to be stable when the NCA was generating them, but only once a structure was finished. Also, Unity's implementation of *FixedJoints* were useful for attaching the edges to the vertices and have them break at specific force thresholds.

What was also gained from using a game engine, was the availability of multiple other inbuilt functions to optimise the environment like object pooling, debugging tools and the availability of the unity asset store. Also unity allows for the increase of its time scale in the physics engine, thereby allowing the simulations to run even faster.

Neural Network & Structure Representation

The first idea for how to represent the structures was as a graph. The structure is essentially just a graph of vertices and the edges between them. Unfortunately, this simple representation proved to be infeasible for generating structures in 3D space, as the representation completely disregarded any idea of spatiality. A vertex directly next to another vertex in space, would not be conveyed to the NCA as the representation would only allow you to know which vertices was connected to each other, and not where in space they would reside. This would make it difficult for the NCA to learn to connect nearby vertices together. Therefore this approach was abandoned.

To include spatial awareness, the area in which the structure was to be built was divided into a discrete grid with a constant specified length between the center of each cell in the grid. To represent the structure in this grid a representation was devised that had 14 one-hot encoded 3-dimensional arrays. One of them would contain the positions of the vertices, and the 13 remaining arrays were needed to represent the connections. This initially seemed like a good representation, and it allowed for the use of a convolutional neural network for the NCA as done by Mordvintsev et al. (2020), where each 3-dimensional array would be one of the input channels. Unfortunately this CNN grew very quickly in size, and even restricting the size of the grid to be a square with side length 50, it still needed to allocate multiple terabytes of memory for storing all the weights and biases. This is due to the fact that the input size of the network would be

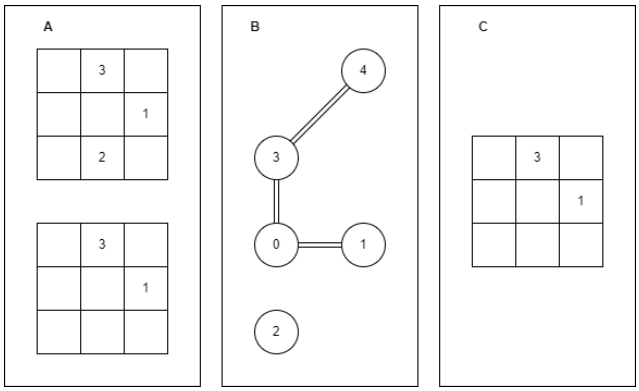


Figure 1: Representation of a 2D structure as a 2D local. The locals are for vertex 0.
Figure A are the two input arrays with vertex positions at the top and connections at the bottom. Notice that vertex 4 is not included here because it is outside the reach of the local.
Figure B is the structure that is represented.
Figure C represent the output structure where only the vertex positions are included as the connections to vertex 0 are implicit. Notice that vertex 0 is omitted in both input arrays and the output array.

1.750.000. For this reason, it was decided to not use this representation.

The final representation for the structures ended up being a mix of the two approaches described above. The structure is represented as a graph, using an adjacency list. Each vertex in the list also stores its position in space. Thereby it is possible to convert from the graph representation to the grid representation, and give this as input to the NCA. When inputting the structure to the NCA, it would iterate over each vertex in the structure and convert it, to what the authors named a *local*. A *local* is a 4-dimensional array that represents a vertex' local surroundings. It is essentially two one-hot encoded 3-dimensional arrays, that each represents a 3-dimensional square grid with side length 3 around the vertex. The first array represents the presence of vertices, and the second array indicates whether a connection exists between these vertices and the vertex whose *local* it is. This representation is what is given as input to the NCA. The output is very similar. The only difference is that the array that represents the connections between vertices have been omitted, and instead it is assumed that every vertex present in the *local*, is connected to the vertex whose *local* it is. An example of a small 2D structure and the input and output locals corresponding to it can be seen in figure 1
Given this representation it has been possible to keep the neural network of the NCA quite simple. The structure of the neural network can be seen in figure 2. It is a fully connected network with an input layer of size 52. This corresponds with the size of a *local*. Two 3D arrays of size 3, with the middle entry omitted in both because the vertex whose *local* it is, is implicit $((3*3*3-1)*2)$. It has an output layer of size 26 as this corresponds to an output *local* as explained earlier. Finally, it has a hidden layer of size 60, which corresponds

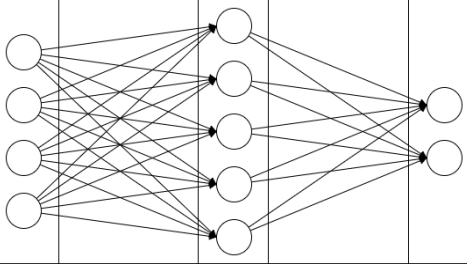
Layer Size:	52		60		26
Network:					
Activation:		Relu		Sigmoid	

Figure 2: Structure of the neural network.

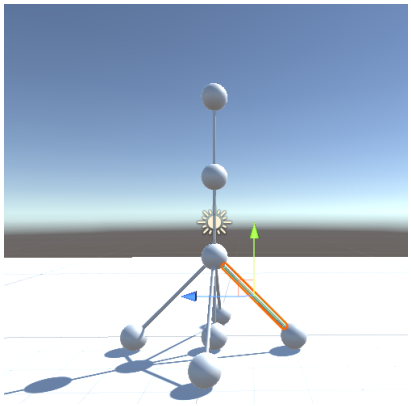


Figure 3: A structure that is 4 vertices tall with a support structure.

to 2 thirds of the input size + the output size. The choice for the size of the hidden layer and the number of hidden layers has been made because these parameters should be good enough for most purposes (Krishnan, 2021). The same reasoning is behind the activation functions. The activation function of the hidden layers is relu, and the activation function of the output layer is sigmoid. Further discussion of the topology of the neural network and the activation function can be found in the section Discussion & Future Work.

For the implementation of the neural network, different libraries was tried, instead of making an implementation from scratch. One version made use of the library TensorFlow (Google Brain, 2021) for python, as this is a well known library that can also run on the graphics card, though since the scripting language of Unity was C#, several layers needed to be added between them to make it work. Even though Unity have added python scripting to new versions of their engine, it would still not work with outside libraries. TensorFlow also has a C# version called *TensorFlowSharp*, however this library did not work with the version of .Net used by the Unity engine. In the final implementation a neural network implementation designed for Unity called Noedify (Tiny Angle Labs, 2020) was used. It had a simple and useful interface that allowed for multiple different architectures for neural networks as well as access to all data stored in the network at any time, which was needed for the neuroevolution algorithm.

Evaluation

In order to evaluate the structures created by the NCAs, and by extension evaluating the NCAs themselves, it was required to be able to simulate whether or not the structures were stable. This was done by using the Unity physics engine, as described previously. The graph representation of the structures produced by the NCAs was converted into Unity *GameObjects*. Each vertex became a sphere and each edge a cylinder connecting two spheres. All with their own *rigidbodies*, so they were able to be affected by the physics engine and would react to collisions. The Unity component *FixedJoint* was used to create the physical link between the cylinders and spheres. The *FixedJoint* component restricts another objects movement to be dependant on another object. This is similar to parenting but is implemented through physics instead of the *Transform* hierarchy. The advantage of using a *FixedJoint* was that it comes with all the properties that were needed for evaluating the structural integrity. One of these properties was the *BreakForce*, which determines how much force the joint can sustain before breaking. For example at a *BreakForce* of 52.5, which was the force used in the project, a structure four vertices tall would break under its own weight, but if supported properly as shown in figure 3, the force on the lower links was alleviated by the support and could therefore stand without breaking. This value was found by building different structures that was either expected to stand or fall, and then the simulation was run with different break forces, until one was found that gave the expected results.

The idea behind the evaluation with these components was to simulate the force of gravity being applied to the structure for an appropriate amount of time in Unity. Then evaluating how many of the joints broke. However, more metrics than structural integrity were necessary to create interesting structures, since a maximum score in structural integrity could be reached by a trivial case of two vertices connected by a single edge. Therefore two more metrics were added to the evaluation.

One metric was determined by the height of the structure, so the NCAs would get better results if their structures were higher after the physics simulation was completed. This should yield a better evaluation score, compared to a completely flat structure, that might have no broken joints. In the implementation this was calculated as the difference between the highest and lowest vertex along the y-axis of the world coordinate system.

The other metric sought to capture the complexity of the structure and was used in an attempt to make the NCAs create more complex structures. Different calculations was tested for this metric, that were all dependent on the amount of vertices and edges and the difference between them, but in the final version it turned out a simple count of how many vertices the structure used provided a fine complexity metric for the evaluation.

To calculate the final evaluation score for a structure, each

of the three metrics discussed above were used in the calculation. Weights for the height and complexity metrics was also introduced, to allow for more control during testing of how much they should impact the final result, in the final implementation both of these values were set to 1. The formula used in the end was:

$$Q = I + (W_h * H + W_c * C) * I$$

Where I is the integrity metric, H is the height metric, C is the complexity metric and W_h and W_c are the weights for height and complexity respectively.

By scaling the height and complexity metrics by the integrity metric, the NCA has a higher focus on creating stable structures, and only once the integrity reaches a certain level will height and complexity be relevant. This scaling was introduced because the NCAs otherwise would completely disregard integrity, and simply just try to maximise height and complexity, which results in creating as many vertices as possible.

Evolution

Backpropogation was quickly dismissed as a possibility, and therefore neuroevolution was used instead to train the NCAs. Since only the weights and biases of the neural networks and not the topology should be evolved, a direct representation was chosen for the genome (Floreano and Mattiussi, 2008). The following sections will detail the implementation of recombination and mutation of the neural networks.

Population and Parent Selection For the evolution population, a size of 100 with a parent size of 20 was chosen. This meant that the population would consist of 100 randomly weighted neural networks at the beginning. Then, for each evolution the 20 networks with the best evaluation score was kept and the other 80 discarded. After selecting the parents they would be used as basis for the recombination and mutation algorithms. As opposed to discarding the parents for more mutations and recombinations the parents was kept for the next generation, only mutating and recombining 80 new networks.

Recombination The recombination algorithm saw a lot of iterations. At first an attempt to use the biases of one parent and the weights of another was dismissed, since it allowed for almost no permutations of the parents when there were only a single hidden layer. Another attempt was to only take half of the biases of one parent and combine with half of the biases from the other parent and likewise for the weights. This approached seemed more in line with the general idea of recombinations, since it takes half the genes of a parent and half the genes of another. However, this had the same result as simply randomising the neural network. Finally, the solution that was the most promising one, was to pick two parents, then pick a neuron in the hidden layer and swap it around between the parents (as well as all its associated weights and biases) creating two new children per swap.

Although this seemed more promising at first, once the other hyperparameters were tuned, this approach also proved to be almost no better than generating completely random networks. It seems even the switch of a single neuron had a vast impact on the entire network. A solution could have been to only swap some weights of a single neuron between two parents, but this became closer and closer to mutation, and thus, recombinations were abandoned in the final version.

Mutations The mutation algorithm is implemented as described by Eiben and Smith (2013), where selected parents produce stochastic modified version of themselves. In the case for the neural networks presented in this project, there was a focus on creating an algorithm that could mutate any fully connected feed forward network regardless of input size. This resulted in an algorithm that could change the weights and biases of a parent in arbitrary layers, but disregard the rest of the parents architecture.

When the algorithm runs it selects a random network from the parents for mutation. Then it selects a percentage of the biases in each layer and modifies each of them by a stochastic value, found within a given range. The range of the modifications of the biases has been set to a maximum range of $[-0.25; 0.25]$. This range was set, since it was the same range used to set the biases for randomly generated networks in Noedify, so if networks were set to a maximum percentage and modification range, it would be equal to generating a completely new network. The same process was used for the selection and modification of weights in the networks, the only difference being that in the neural network's representation of the weights, were implemented as two-dimensional arrays.

Results

The following section will go over the results of the experiments that have been run and explain how the hyperparameters were tweaked to try to reach the best performance possible.

All experiments were done with a maximum of five iterations. This means that the graph was run iteratively through the NCA five times, before the structure was built in Unity and then evaluated. The reason five was chosen, was that if the iterations were set too low the latent space of possible structures to create became very small, which makes for uninteresting results. On the contrary, with too high maximum iterations, the latent space becomes very large, and the simulation time of the structures slows down the evolution quite a bit. As such the perfect balance was reached with a maximum of five iterations.

The largest test that was run was a 40 hour experiment with a population of 100. The experiment produced very predictable results and helped confirm speculations like the evaluation function being adequate. As previously mentioned, the evaluation function multiplies the height and complexity with the integrity metric which is normalized. This means that if the structure is not structurally stable, it will not matter how complex or high it is. Therefore the hypothesis was that the evolution would gradually create structures with a higher and higher integrity score until it finally reaches a score of 1, at which point it will begin expanding the structure as well as building upwards to maximise both height and complexity. This was what happened and is

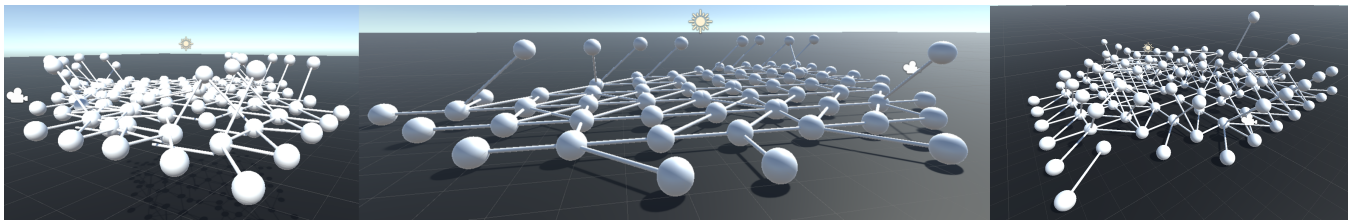


Figure 4: From left to right: Spawn number 6135, Spawn number 7480, and Spawn number 28577.

shown in figure 4.

It starts by gradually increasing the integrity. On the first picture (from left to right), spawn 6135 is seen, which is close to reaching an integrity of 1, but still has some broken joints due to some of the structure falling when being simulated. On the next picture spawn 7480 is seen, the first member of the population to reach an integrity of 1. Here it can be seen that it has created a strong foundation, and can now start to build upwards and outwards without the integrity negatively affecting the height and complexity metrics. Finally, spawn 28577, the final structure of the 40 hour test. Not only has the structure started building upwards, but it has also connected many of the vertices in the higher layer, which gives it the structural support it needs to keep building upwards. This was even conceptualised by the vertex sticking up in the back of the picture, showing that it was ready to start creating vertices on the third layer.

Benchmark

This section will look into the runtime of the different parts of the implementation and describe the largest bottlenecks. One of the problems with the evolution algorithms was that iterations were very slow. When the integrity was less than 1, it was often because the structure was quite large and had so many vertices that it crumbled under its own weight. As mentioned before, in the large experiment, an integrity of 1 was reached at spawn 7480, which was created in the 93rd evolution iteration. This means that up until about the 100th iteration, the NCAs are creating enormous graphs that have to be simulated, which was the biggest bottleneck when running the evolution.

In table 1 the results of the benchmark tests are shown. As can be seen the Evaluator, which runs the physics simulation, is by far the part of the algorithm that takes the most time. It is also evident how big a difference the average integrity of the population plays. Going from 2044 seconds to 131 seconds as the integrity increases. It is important to note here that it is not the integrity per say that makes the difference, but the fact that an initial high integrity is synonymous with smaller structures. And the fewer vertices a structure has the faster it is to simulate.

Special rules

As can be seen in table 1, once an integrity score of 1 is reached, much smaller and self-contained structures are built. This means, that after this point the speed of the simulations increases significantly. Therefore experimentation

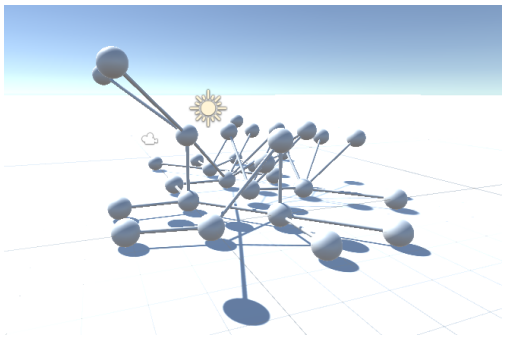


Figure 5: Evolution with activation cutoff at 0.6, Spawn number 381.

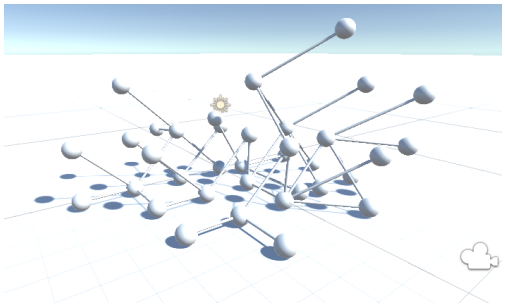


Figure 6: Evolution with activation cutoff at 0.6, Spawn number 5800.

with adding rules that could make the NCA reach an integrity of 1 faster was initiated. The first experiment was to increase the cutoff rate, for when an output neuron is considered active. This helped by lowering the initial amount of vertices in the random networks and as such greatly increased performance, both in speed and in evaluation. With fewer vertices it was possible to reach structures with an integrity of 1 faster. Some of the results from these experiments are shown in figure 5 and 6.

As shown in figure 6, the NCA reached a height of four vertices, which, as was explained earlier, was only possible, if there was a supportive structure that holds the ”tower” with more than one vertex. The problem with changing the activation cutoff was that, much like lowering the maximum iterations, it shrinks the latent space as it puts a restriction on the number of vertices.

Another rule was to not allow the NCA to build down-

	Neural Network Build Average	Unity Graph Build Average	Evaluator Average	Evolution	Total runtime for 1 iteration
Average Integrity <0.5	1.46	0.48	25.06	1.21	2044.41
Average Integriy >0.5	0.98	0.28	11.99	0.41	1060.41
Average Integrity = 1	0.20	0.02	1.42	0.05	131.25

Table 1: Table of benchmark test results measured in seconds. Each iteration have been done with a population size of 100, parent size of 20, 30 seconds of physics simulation and five iterations through the NCA per structure.

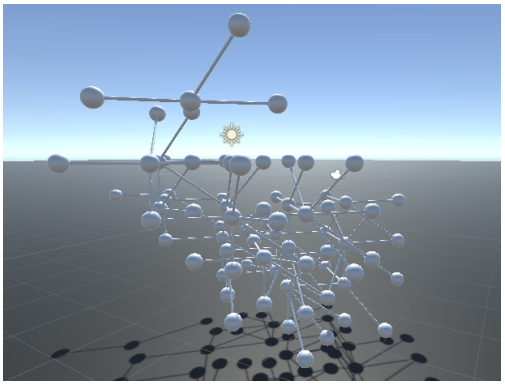


Figure 7: A structure that is about to fall down when simulated. The center of this structure has y position 0.

.

wards. This was a global rule, and breask the idea of using an NCA. The rule was that if the network wants to place a vertex beneath the global y coordinate of 0 it was disallowed. This means that it was not only looking at locality but also the vertex' global position. The reason for experimenting with this rule, was that when the network tried to build downwards, it would often build very unstable structures, that might have been fine, if it hadn't build down such as the one shown in figure 7. Here the center of the structure has y position 0, meaning that it would probably have been a stable structure, if it had not built downwards.

In figure 8 a structure was created with the "no build down" rule. An interesting thing to note, is that this structure looks a lot like the one created at the end of the long experiment. The difference here is that the structure from the long test had spawn number 28577, while this "no build down" structure has the spawn number 6811. So it requires almost 4 times fewer iterations to reach a similar result to the long experiment by introducing this rule. However, as should be evident by introducing a rule that removes vertices if they cross some arbitrary rule, is that the structures it generates contains far fewer vertices than the structures from the long experiment.

Average Height

Another experiment was to change the height evaluation to look at average height instead of total height. The idea was that since structures higher than 3 requires more support, this can be achieved by having 3 towers of height 3 and connecting them all to a fourth tower at height 4. In order to do so, the NCA needs to evolve towards a structure that has more than one point in max height. Therefore the idea of evaluating with average height was introduced. The result as seen in figure 9, is not what was aimed for. The NCA simply learned that by creating a small foundation and then building up it could achieve a really high average height evaluation score.

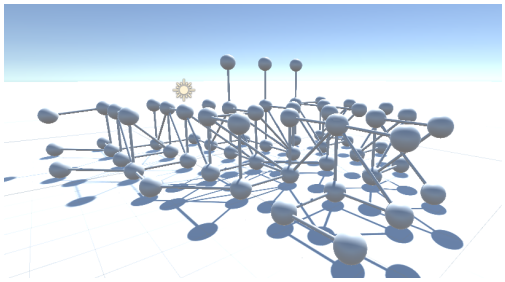


Figure 8: Structure build with "no build down" rule. Spawn number 6811.

.

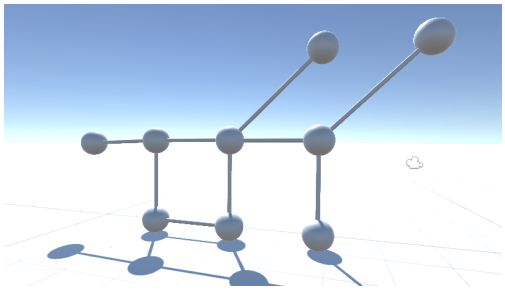


Figure 9: Structure evolved with the average height evaluation.

.

Summary

To summarise, the NCAs produced with the best performance in terms of integrity and complexity was by far the ones produced by the 40 hour experiment. Though great improvements to performance in terms of runtime complexity could be achieved by tinkering with things such as global rules and different activation cutoffs for neuron activation, it seems that letting the NCA evolve with no guidance but the fitness function produces the best results. However, even with a 40 hour experiment the structures never reached a

height of 4 vertices, which was something that was achieved in the other experiments with a higher activation cutoff. This could be explained as a tradeoff between the complexity and the height, since when you have more vertices you have a higher risk of your upwards advancement to have an unintended negative effect in other parts of the structure. Therefore, structures that are smaller and less complex, will have an easier time evolving towards a larger height.

Discussion & Future Work

This section will further discuss the implementation details from the previous section, looking into alternate solutions for the implementation choices that were made. It will also reflect upon what the future work of the project would entail.

Learning Strategies

This project has been using an evolution strategy to create networks, which has had some effects on the NCA. By using an evolutionary search algorithm on a population of networks instead of training a single network, multiple networks had to be created which all took time to create, run and destroy. One of the problems with the evolutionary strategy is to determine appropriate values for recombination and mutation. Both of these can result in what behaves as new random networks if their parameters have been set too high, rather than using learning to change the network a little bit over time. And as explained earlier, a recombination algorithm that didn't result in this behaviour was never implemented.

It is worth considering other learning strategies, as there are multiple other techniques to train neural networks. One technique that could also have been used would be supervised learning with backpropagation, where the network would be trained on data with known results and compare this to the prediction made by the network (Han et al., 2012). This would require being able to calculate diffenrentiable loss functions. Since there is no way to determine what the best outputs for a given input should be, as this is exactly what this project was trying to discover, it would not be possible to use supervised learning.

Another strategy that works quite similar to the evolution is reinforcement learning, in this environment the network would be able to make a decision on where to grow vertices and edges and would then get an instant response depending on the current policy (Yannakakis and Togelius, 2018). It would be interesting to study the differences between using this technique with step-wise evaluation and reward as opposed to the one implemented where only the final structure is evaluated. Though a proper policy needs to be created to achieve the same result with higher structures that are not necessarily symmetrical. This comparison would be one of the next steps of the project.

Finally it should also be noted that the cost of generating and destroying the numerous neural networks was insignificant, compared to the cost of simulating the force of gravity on the structures.

Topology of Neural Network

All decisions regarding the topology of the neural network was based solely on the common rules regarding such. These rules are based in part on the authors' own previous experiences and knowledge of neural networks and in part on and article by Krishnan (2021). This is also the case for the choice of activation functions.

Due to the large amount of hyperparameters present in the implementation (population size, mutation rate, structure representation etc.), and the limited time scale of this project, it was decided not to focus on any hype parameters of the neural network to reduce the number of different permutations of all of these. Instead the focus was placed on the hyperparameters of the evolution algorithm, evaluation strategy and structure representation to limit the scope of the project.

Experimenting with the neural networks hyperparameters is a very obvious next step for the future work on this project. Testing different number of hidden layers with more or fewer neurons, and different activation functions could have a large effect on the different kinds of structures the NCAs are able to generate. It could also have been interesting to study the effect of disabling parts of the network, to see if that would show better results. An approach similar to dropout, where single neurons have a chance to be disabled and the rest of the network still has to work without them (Hinton et al., 2012). The interplay between this and the evolution algorithm would also be interesting to study, as this would introduce some stochasticity to the evolution.

It would also be interesting to let the evolution algorithm also evolve the topology of the network. As stated previously the algorithm only evolved the weights and biases of the network. By also allowing the evolution to change the topology of the network, it could possibly also find even better networks than the single version that was designed for this project.

Recombination

As described previously in the implementation section, different attempts were made to write a recombination algorithm, that could work on neural networks. The failed implementation attempts shows that recombination was almost just like randomly creating new networks. This behaviour is to be expected when making such drastic changes to a network, as neurons in a fully connected network would be very dependant on each other and different neurons would learn different things. A project for the future could be to study this aspect of recombination of neural networks in more depth, to see if it could be possible to make it work properly.

Physics Engine

The biggest bottleneck of the runtime cost of the implementation was the physics simulation. This is in part because the simulation would have to run for a sufficiently long time, such that the it wasn't ended prematurely. Otherwise this would result in false evaluation scores for structures that broke slowly. It is also a result of the physics engine in unity

becoming quite slow, once a significant amount of vertices and edges were present. Also, it was impossible to run the simulations in parallel.

Even though Unity has the possibility of increasing the time scale of the simulations, which was done, this was still the biggest bottleneck.

Alternate solutions would include using a simpler physics engine which only simulates the physics effects needed, and thus should be able to run faster, and maybe in parallel. It might also be possible to calculate the integrity of the structure using only mathematics, thereby removing the need for any simulations, and in turn almost completely diminish the runtime of the evaluations.

The reason the mathematical approach wasn't used was because none of the authors are familiar with that area of mathematics, and the short time scale of this project didn't allow time to become familiarised with the subject. Secondly, it seemed like a good idea to actually use a physics environment for the evaluation since one of the long term goals of this project is to make the generation of these structures useful for procedural content generation in games. Therefore it made sense to evaluate in the kinds of environments they would end up in. As for the choice of the Unity engine as the physics environment, this was mostly because this was a physics environment the authors were already very familiar with, and allowed the focus to be on the actual implementation of the NCAs. Had the restrictions of parallelisation and the general runtime cost been known, another physics environment would have been chosen.

Conclusion

This paper has shown that NCAs, trained via an evolutionary search algorithm, are capable of generating 3D structures that are stable under the force of gravity. Though much more work needs to be done before it can be applied in for example procedural content generation, it is an important first step. It seems very likely that further development of this project could result in impressive structures that would be useful in different virtual 3D environments. It also seems likely that this project can be tweaked to not only generate stable structures, but also structures with other traits, simply by changing the evaluation function. Hopefully this paper can serve as a small expansion of the current understanding of what NCAs can be used for and open up the path for further research into the generation of stable structures.

References

Earle, S., Snider, J., Fontaine, M., Nikolaidis, S., & Togelius, J. (2021). Illuminating diverse neural cellular automata for level generation.

Eiben, A. E., & Smith, J. E. (2013). Introduction to evolutionary computing. Springer.

Floreano, D., & Mattiussi, C. (2008). Neuroevolution: From architectures to learning. *Evol Intell*, *1*.

Gardener, M. (1970). Mathematical games the fantastic combinations of, john conway's new solitaire game "life". *Scientific America*, *223*, 120–123.

Google Brain. (2021). Tensorflow. https://www.tensorflow.org/

Han, J., Kamber, M., & Pei, J. (2012). Data mining concepts and techniques. Elsevier.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors.

Krishnan, S. (2021). How to determine the number of layers and neurons in the hidden layer? [Accessed: 2021-16-12]. https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3

Mordvintsev, A., Randazzo, E., Niklasson, E., & Levin, M. (2020). Growing neural cellular automata: Differentiable model of morphogenesis. *Distill*, *5*.

Sudhakaran, S., Grbic, D., Li, S., Katona, A., Najarro, E., Glanois, C., & Risi, S. (2021). Growing 3d artefacts and functional machines with neural cellular automata. *Proceedings of the 2021 Conference on Artificial Life*.

Tiny Angle Labs. (2020). Noedify - easy neural networks. https://assetstore.unity.com/packages/tools/ai/noedify-easy-neural-networks-161940

Unity Technologies. (2005). Unity. https://unity.com/

Yannakakis, G. N., & Togelius, J. (2018). Artificial intelligence and games. Springer.