Procedural Content Generation via Genetic Algorithms

Kevin Altieri
New York University
New York, USA
kda285@nyu.edu

Tarek Hassoun
New York University
New York, USA
th1750@nyu.edu

Alexander Graham
New York University
New York, USA
ag5278@nyu.edu

Abstract—**This project focuses on expanding upon the new approach to Procedural Content Generation through the use of Reinforcement Learning. With the recent growth in attention towards PCG, many older styles of content generation such as search or solver methods are being replaced with different approaches such as Reinforcement Learning, L systems, and more. These approaches, while showing excellent results, are mainly policy-gradient based, and can be pretty heavy to run. We investigate a gradient free approach based on neural networks and genetic algorithms. Our method works by building a neural network, training it by having it modify GVGAI game levels, and getting the best chromosomes through the genetic algorithm. Once done, we take a look at how well the best chromosome can build a playable level from a randomly generated one. We additionally check overall how the method runs by looking at the top overall average reward. This work is done to compare to similar experiments run with reinforcement learning methods.**

I. Introduction

Procedural Content Generation (PCG) is a method of creating content algorithmically as opposed to doing it all manually. While more work needs to be done, the overall objective of PCG is to provide the game industry with a cheap tool that can produce unique and qualitative content at a fast rate. Recently, interest in PCG has picked up, with new ideas emerging such as using Reinforcement Learning in place of traditional evolutionary algorithms. Currently, the RL approach to PCG focuses on content creation as an iterative process rather than a single complete process. When Reinforcement Learning (RL) is introduced into the formulation of PCG, level design is framed as game content that the RL system can use as learning policies to determine the most efficient "next step" to maximizing the output quality of the PCG system. There are many advantages to this approach such as cost reduction, faster level generation, no training data, and the fact that trained level policies that are maintained incrementally have a tendency to work really well with diverse and mixed-initiative approaches that PCG is known for.

Artificial Neural Networks (ANNs) or simply neural networks, are computational models that attempt to model the human brain. A neural network is a set of processing units (neurons) that take in inputs and give them an associated weight. The input weight totals are then added to a bias, which is fed to a function that gives an output for that neuron [1]. A neural network is a directed graph of individual neurons, with the first layer and last layer being called input and output neurons respectively, with the in between layers being called hidden layers. Neuroevolution is when a neural network is used in conjunction with an evolutionary algorithm. Neuroevolution is considered to be quite flexible, due to its ability to approximate general underlying functions found in most AI problems, in addition to being grounded in biological modeling (as seen above). The strategy also gives the advantage of not requiring input output pairs, instead only needing a measure of a network's performance to train [2]. In our model, this measure is the total average reward gained from playing games. The general pattern for a neural network will be explained in further detail below, but in general involves initializing chromosomes that are encoded into the neural network, to optimize some fitness function that shows the quality of the network. The chromosomes are tested, and selected for a new population of chromosomes, which is done through mutation or crossover. Lastly the new chromosomes are placed into the population, replacing some old ones. This process repeats as many times as necessary.

In this paper, we plan to discuss our approach to procedural generated content, specifically for creating levels, using our genetic algorithm.

II. Related Work

The foundation of this research lies heavily on the previous research done in using Reinforcement Learning as a means to improve the efficacy of Procedural Content Generation. The work in question that we are relating is PCGRL: Procedural Content Generation via Reinforcement Learning [3] where they explore the advantages and challenges that come with the implementation of Reinforcement Learning within a PCG based architecture. The PCGRL framework is unique in that it casts the PCG process as an iterative task instead of a complete process. Their method treats content creation more like a game, and by doing so teaches the agent how to generate content. Their model is based on using a neural network, along with Policy Proximation Gradients (PPO), a type of actor critic policy. They then feed into their network the game information, and in return get the reward, observation of the game state, and value, which is used in the PPO algorithm. Additionally, the PCGRL team uses GVGAI games as their problem set, and a variety of different observation spaces as their representations.

While the PCGRL framework is excellently done, Reinforcement Learning in general can be quite time consuming. Evolutionary Strategies are another option for content generation as they often do a good job competing with reinforcement learning algorithms on content generation as well as other fields. However, in a paper penned by Uber AI Labs, they focus on a neural network with a genetic algorithm, moving away from policy gradient methods (which they consider backpropagation evolutionary strategies to be). Their genetic algorithm is basically the default genetic algorithm, where a population of individuals has a genotype made up of the network parameters, that is evaluated each generation, with the best individuals going on to be parents for the next generation. Two differences in their algorithm was first the exclusion of a crossover event, which they removed to simply their experiment. The second was in efforts to make their algorithm more efficient. To increase efficiency and storage, the Uber team developed what they called "distributed deep GA" where they stored the parameters by representing each as an initialization seed, along with the list of random seeds that produced the mutations that produced each individual [4]. This allowed them to increase the number of agents, and keep the distributed list independent of the overall system - giving them a huge amount of compression. The team then ran their deep GA against many other reinforcement learning and ES algorithms such as DQN, and AC3. Overall their data shows that for their model, which they ran on either equal or more frames than their counterparts often finished days sooner, with roughly the same number of games won.

The Uber paper and Evolution Strategies as a Scalable Alternative to Reinforcement Learning [5] show that Evolutionary Strategies can be highly competitive with Reinforcement Learning methods, sometimes even improving on them. This is due to a number of advantages that ES has over RL algorithms. Firstly, ES is highly parallelizable compared to RL which is mostly due to the fact that ES only requires the chromosomes involved to communicate few scalars between one another. RL on the other hand, needs to synchronize entire parameter vectors or weights in order to go through parallelization. ES is also extremely robust, which means it can be scaled upwards very easily, whereas RL is forced to add on more hyperparameters which can be difficult to implement in order to realize that scale. A lot of RL algorithms initialize with random policies which can lead to unfavorable outcomes such as being stuck on a local maximum / minimum instead of going towards their global counterparts. Even if this effect is less apparent in RL implementations like Q-learning, ES does not suffer from these problems because deterministic policies create an environment where there is diverse and consistent exploration of the action space. Finally, the biggest advantage that ES has over RL is that there is no need for backpropagation. Changing the weights in a structured manner such as backpropagation is a must for Reinforcement Learning architectures, but is not at all mandatory for many Evolutionary Strategy architectures. In fact, ES only requires the forward pass policy before performing its gradient step, which can (in most practices) make the code a lot shorter and noticeably faster. Past episodes don't need to be recorded in ES, so in addition to an improved runtime we're not troubled with the possibility of running out of memory unless the initialized chromosomes were large in size to begin with. It is all of these possible advantages in mind that we selected to create our own neuroevolutionary algorithm to compare to the reinforcement learning methods used in the PCGRL experiment.

Procedural Content Generation also is gaining popularity in multiple genres of the computer game industry. Platforming games have a history of causing issues for PCG due to the unstudied science of level design and a weak understanding of what makes a level successful. In the paper A Genetic Approach in Procedural Content Generation for Platformer Games Level Creation, researchers from the University of Kerman attempted to create a PCG for platforming that goes beyond the basic standard of stringing together pieces of old human designed levels [6]. They used a genetic algorithm to evolve quality levels from a starting population of very simple, short levels. The complexity of the levels was categorized and generated by a metric that they measured for dexterity-based challenges called rhythm. Rhythm involved creating varying patterns of repetition and rhythmic-timing for distinct user actions. Creating a grammar of user actions allowed the researchers to generate varying difficulty by searching for different lengths and strings of required user action based on how the level was designed. By mutating the rhythm for PCG, neuro-evolution was able to generate interesting levels. Another team created a game called PCG: Angry Bots [7]. They attempted to create an action shooter which generates levels targeted at the player's skill level. After a level is completed the user reports back a score between 1 and 6 to determine user enjoyment. The levels are then generated in a randomized space and filled with different types of enemies and pick-ups such as health or ammo. Instead of a standard reinforcement learning, they utilize a Neuroevolution of Augmenting Topologies algorithm. As more PCG is proven successful when paired with neuro evolutionary algorithms, we became inspired to test a genetic algorithm's efficacy.

III. Methods

Our methods are based on the PCGRL GYM interface, which itself is built on the OpenAI GYM environment. The GYM interface is itself a toolkit used for developing and comparing different reinforcement learning algorithms. This is done through a variety of games and models that not only come with the toolkit but can also be added on by the user through different classes and scripts. In the original implementation, a Stable Baselines training method along with Proximal Policy Optimization (PPO) is used to establish a convolutional neural network and train agents. In the binary representation, which is what our method focuses on, the body consists of 3 convolution layers followed by a fully connected layer, and then a final fully connected layer for both the action and value heads. This training, forwarding, backtracking, and action sequence via this convolutional neural network is run for a certain number of frames (100 million) or up until the agent is finished constructing the map [3]. For each problem / representation experiment this is where the similarities between our project and the original PCGRL GYM deviates. Our model focused on building a neural network with a genetic algorithm using Pytorch instead of Stable Baselines. In order to apply our neuroevolutionary strategy to the GVGAI framework, we had to remake the convolutional neural network, while maintaining the same input and output channels used in the PCGRL experiment [3], [8]. We start off with 3 convolutional layers that apply a two-dimensional convolution over an input signal of several input planes. Then we proceed with a fully-connected Linear layer, which applies a linear transformation:

$$y = x * A^T + b \tag{1}$$

to the incoming data. The linear layer's input channel is defined to be $map_size * map_size * output_of_conv3$. Our final layer is what we refer to as our pilogits layer, a fully connected layer that applies the same linear transformation. Pilogits represents the probabilities for the different actions our agent can take. Our inputs, which are Tensors that contain matrices within them, are transformed at each layer to be semi-orthogonal. On every forward pass, all layers go through a functional ReLU transformation, where weights are changed to follow

$$x = max(0, x) \tag{2}$$

Our pilogits layer does not go through the ReLU transformation but instead becomes the action set that is returned to our system. This action set is represented by a torch Categorical object, which are categorical distributions that are parametrized by either probabilities or, in our case, logits.

Our Genetic Algorithm is roughly similar to that used in the Uber Labs experiment, with some things moved around. Instead of initializing the chromosomes in the first generation, we separate the generation running and the initialization; we also do away with the Elite candidates used in the paper [4]. In initialization, we prepare the chromosome's GVGAI game environment, and the change percentage to 40 percent, following the work done in the PCGRL paper. Additionally, every chromosome also holds its own genes, which is a one-dimensional array that is the flattened size of all the weights and biases of each layer in our convolutional neural network, which are based on the network's weights and biases. In the advancement process, we make each chromosome run its fitness method in order to establish its fitness value, which will be the average of the total reward accumulated over the number of episodes it must go through.

The fitness method starts by re-distributing the data within the chromosomes genes back to the neural network (in the same order of course). This means whatever modifications have happened to our gene pool get properly re-distributed to the weights and biases of each layer in our network. Then for each episode loop, each chromosome's environment gets reset in order to obtain an observation. This observation is turned into a Tensor object, which is then used to obtain an action from the neural network forward pass. This action is then used as a "step" in our environment object, and a reward is returned for us to calculate the total accumulated reward. At the end of all the episodes, we return the fitness as:

$$fitness = totalReward/numberEpisodes \tag{3}$$

After fitness has been run for each chromosome, we proceed to sort our chromosomes based on their fitness value from most successful to least successful. The fitness average of the successful chromosomes is recorded in a separate variable for graphing later-on. At this point we use a mix of the Uber Labs and the Canonical ES method to do crossover [4], [9]. At the point of crossover we make use of two variables that are defined on initialization by the user, μ (mu) and λ (lambda). The first of these variables, μ , is responsible for picking the number of chromosomes for crossover. Since we sorted our population, this will be defined as the first n best chromosomes. These chromosomes, defined by μ, create the offspring crossover via a copy method rather than a gene splicing method. So the first n best chromosomes are each copied once to create n children. These children then undergo mutation via a mutation process that guarantees that at least 10 of their genes will be affected by a random Gaussian Distribution value with mean of 0 and standard deviation of 0.1. Finally comes the integration of these children into the population, defined by λ.

Here, as we progress from start to finish, is used for population insertion. At the start of our crossover / mutation process, children will go ahead and start by replacing the parents who under performed. Later children will simply add-on to the newly formed population without replacing any chromosomes. The population outcome is highly based around the definition of (mu) and (lambda), and given that we want results that can be properly evaluated without ever diminishing or breaking our population, we need to go ahead and make sure that $TotalPopulationSize > \lambda \geq \mu$.

The above processes are run for a certain number of generations per Genetic Algorithm object and we evaluate performance by looking at the average fitness levels over time of the highest performing chromosomes. We also save the state of the best neural network in each run, to be used to create rendered maps of the network playing in the game environment once trained. Our hope is that, with enough computation power, a neuroevolutionary approach can outdo and even improve upon the standard GYM-PCGRL implementation. Below is a formatted example of our genetic algorithm, as we have explained above:

Algorithm 1 Our Simple Genetic Algorithm

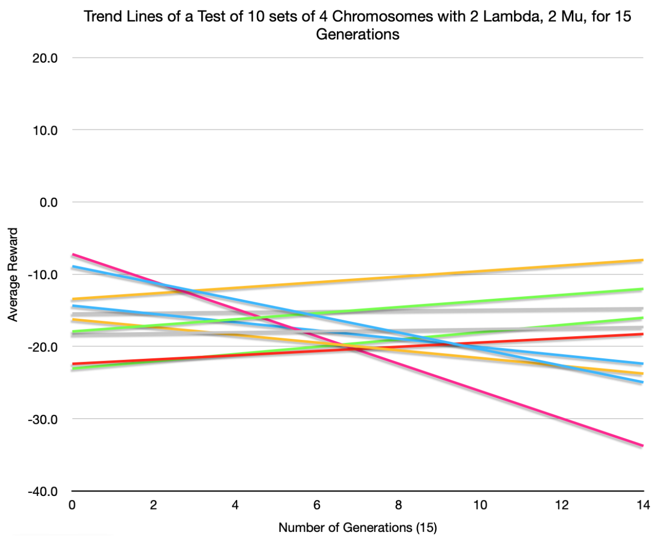
```
Input: C number of chromosomes, Input: E number of episodes, $\mu$ for mutation, $\lambda$ for replacement, G number of generations
initialization: for c = 1, 2, 3 ..., C chromosomes do
    c.net = Initialize CNN
    c.genes = CNN.weights
end
generation: for g = 1, 2, 3 ..., G generations do
    foreach chromosome do
        Update c.net weights for e = 1, 2, 3 ..., E episodes do
            obs = new environment while not done do
                action = net.forward(Tensor.obs)
                reward = env.step(action)
                totalReward += reward
            end
        end
        $fitness = totalReward/E$
    end
    Sort $(\epsilon_1, ..., \epsilon_n)$ chromosomes according to fitness
    for $u$ in $\mu$ do
        child = parent
        modify genes $= random(child.genes/10)$
        foreach gene in modify genes do
            gene += Gaussian Noise ($N$(0, 0.1, 1))
        end
        $pop[\lambda + u] = child$
    end
end
```

IV. Results

Due to the limited resources at our disposal, in terms of both time and computing power, we decided to focus on smaller scale tests that provide a sense of what our algorithm could do. As such, our tests will not compare to those run in the PCGRL experiment, as doing so would not produce objective results. Additionally, we decided to focus on the binary problem, which consists of 3 possible actions (stay, change to empty, change to wall) [8], with the narrow representation, which represents the map with a 2 dimensional array, the position with a simple 1 dimension x, y and a discrete space that represents the value [3]. Most of the tests were run using 4 individuals, with 2 μ and 2 λ. The number of generations vary based on the test.

A. Testing The Fitness Function

This first test was run to see if overall the fitness function was working correctly. In Figure 1, we graph the overall trend lines of the best fitness per generation. We ran the small default sample of 4 individuals with 2 μ and 2 λ. The top graph is from our first series of tests, and the bottom one is from the second series of tests. In the top graph, more of the trend lines are gradually moving upwards, while in the bottom graph the number of trend lines going upward is about even with the number of lines going downward. With the parameters we set, the average fitness usually started around -10 to -20. It



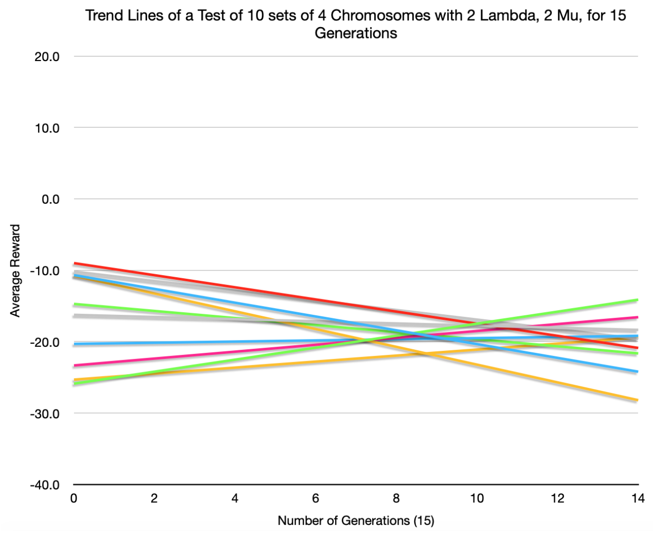
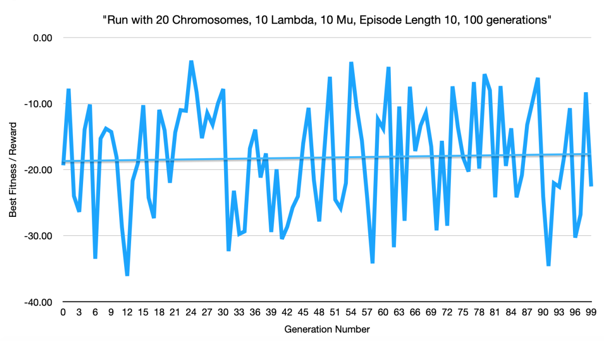


Fig. 1: Trend lines of 2 different tests of running small number of chromosomes

was also interesting that the left graph had values that were much closer together than the right graph. These observations could be from a number of factors, ranging from the test size, to the episode count, to the number of generations. While we were confident enough to continue forward with our other tests, more testing would need to be done to ensure the fitness function works as intended.

B. Comparing More Generations to More Chromosomes

Our next set of testing was looking at how our algorithm performed with more chromosomes versus doing a longer run with the default smaller set. In Figure 2, we have 2 graphs that show average fitness of the top 10 percent of chromosomes. The trend line for the top graph representing 20 chromosomes appears to trend slightly more upward than the bottom one. While there is not enough data to make a for sure conclusion, it does appear that the more chromosomes there are the better



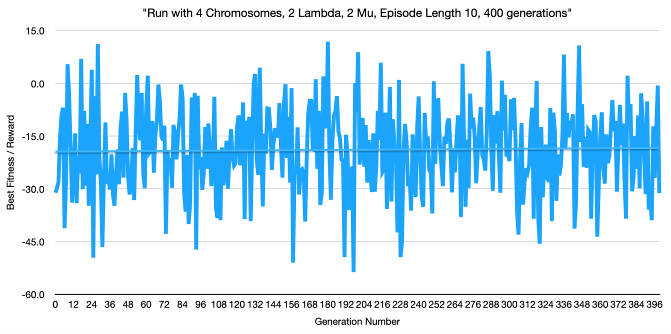
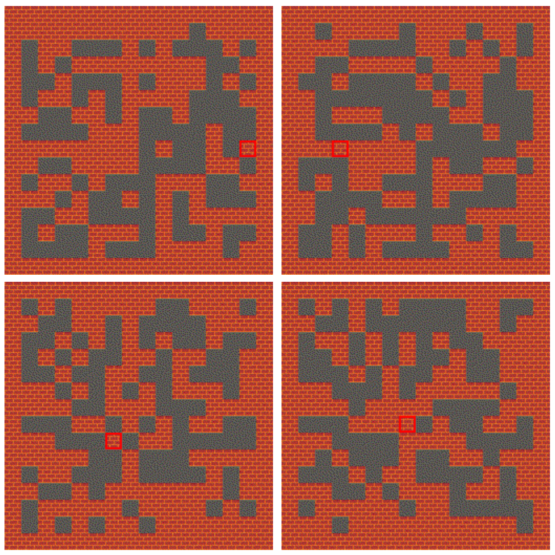


Fig. 2: Reward per generation with trend line for Chromosomes versus Generations test

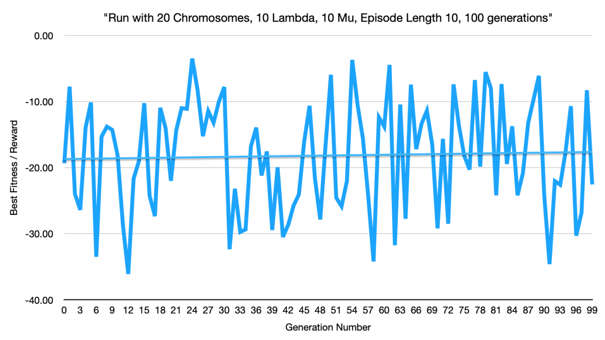


(Initial State) (Final State)

Fig. 3: More Chromosomes versus More Generations Render

the fitness. For this test, we also have a map render of our networks running in the game environment. This is shown in Figure 3. These maps are not successful; a success would have been a fully connected map without any closed off spaces. What they do show is some slight improvement in putting together a cohesive level. Unlike showing the overall fitness, the neural network required to make a successful level is much greater than what we were working with.

C. Comparing Episode Lengths



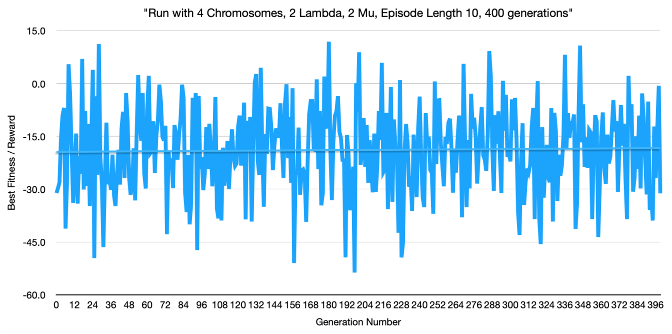
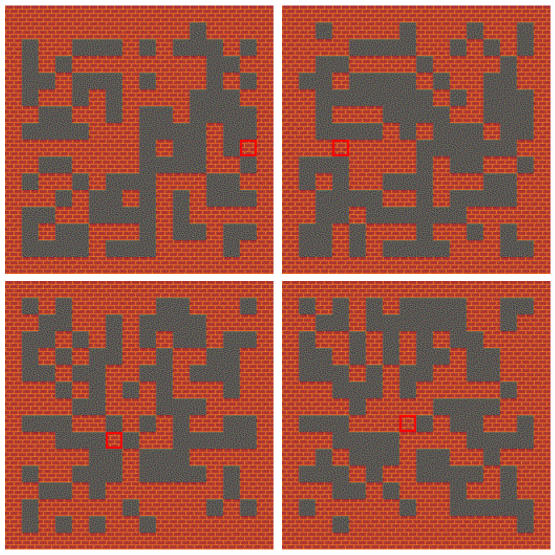


Fig. 4: Reward per generation with trend line for Episode Length Test



(Initial State) (Final State)

Fig. 5: Fitness Function with 10 episodes versus 20 episodes Render

The final test was conducted on the small state of 4 individuals, with 2 mu and 2 $lambda$, but with one having a fitness episode length of size 10, and another of size 20. As seen in Figure 4, the trend lines for both are relatively the same, with the episode length 20 experiment starting with an average fitness of -16, while the average for the episode length 10 experiment started around -20. This could be due to the better fitness function, but based on the data in Figure 1, it could just be random chance. In Figure 5, we have the render maps for the two tests. It appears that the longer episode length may have created a slightly better map than the episode length of 10. However, due to the random generation of the maps, it is possible that the map generated for the higher episode length was more favorable to that network than the one generated for the 10 episode length network.

V. Conclusion

In this paper, we discussed a comparison between reinforcement learning and neuroevolution in the context of procedural content generation, specifically generating levels. We explained the background of the novel procedural content generation using reinforcement learning approach, and how it uses iterative designs and gradient policies to make their work happen. We discussed the benefits and use of neuroevolution in the same space, and how our method operates. In the end, we showed the results of different tests run on our algorithm, and what they could mean for future work.

There is a lot of potential for future research in this area. The immediate future research for this paper would be to perform large scale testing on all of the different problem sets and representations, and compare them to the PCGRL paper. Doing so would either continue to prove that neural networks with genetic algorithms are a viable option, or not. Other more immediate goals would be to further optimize our algorithm. The first would be to find the initial weight values of networks with back- propagation, as results have indicated that networks with evolved initial weights can be trained significantly faster and better than randomly initialized network weights [1]. This is also the method used in the Canonical evolutionary strategy [9]. The second would be to introduce parallelization, to allow our algorithm to take advantage of the better computing power it would be running on.

The long term goals of this paper are similar to the open challenges section of the NeuroEvolution in Games: State of the Art and Open Challenges [10]. That is, neuroevolution is a powerful tool, and one that can be applied to a variety of situations. By testing more scenarios and techniques, hopefully new tools can be created that help designers of all types create high quality, cost efficient procedurally generated content.

References

[1] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," 2008.

[2] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018.

[3] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," 2020.

[4] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *CoRR*, vol. abs/1712.06567, 2017. [Online]. Available: http://arxiv.org/abs/1712.06567

[5] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017.

[6] A. B. Moghadam and M. K. Rafsanjani, "A genetic approach in procedural content generation for platformer games level creation," in *2017 2nd Conference on Swarm Intelligence and Evolutionary Computation (CSIEC)*, 2017, pp. 141–146.

[7] W. L. Raffe, F. Zambetta, and X. Li, "Neuroevolution of content layout in the pcg: Angry bots video game," in *2013 IEEE Congress on Evolutionary Computation*, 2013, pp. 673–680.

[8] D. Pérez-Liébana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General video game AI: a multi-track framework for evaluating agents, games and content generation algorithms," *CoRR*, vol. abs/1802.10363, 2018. [Online]. Available: http://arxiv.org/abs/1802.10363

[9] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "Back to basics: Benchmarking canonical evolution strategies for playing atari," *CoRR*, vol. abs/1802.08842, 2018. [Online]. Available: http://arxiv.org/abs/1802.08842

[10] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *CoRR*, vol. abs/1410.7326, 2014. [Online]. Available: http://arxiv.org/abs/1410.7326