Multi Agent Self-Play Reinforcement Learning for Procedural Content Generation

Cameron Ballard
New York University
New York, USA
clb478@nyu.edu

Dinesh Sreekanthan
New York University
New York, USA
ds5786@nyu.edu

Xianbo Gao
New York University
New York, USA
xg656@nyu.edu

Zaid Baba
New York University
New York, USA
zrb233@nyu.edu

***Abstract*—Recent research has brought about a novel approach to procedural content generation in games by using reinforcement learning. This naturally opens the possibilities of extending this to use different types of reinforcement agents. Therefore, we want to investigate the performance of a self-playing agent for designing levels. We also want to extend the types of levels generated to include multiplayer maps, which seems to be well-suited for self-playing agent design. Adapting the old reinforcement learning agents presents its own set of challenges too, in order to avoid new issues brought about by implementing this. Ideally, this project could serve as an example of the versatility of PCGRL and pave the way for future adaptions of it.**

I. Introduction

Procedural content generation (PCG) in games, if done well, can provide players with a potentially endless stream of content for their favorite games. There is promise in a new type of PCG-using reinforcement learning (RL). A schema for adapting RL problems into the framework of PCG content (PCGRL) has already been laid out [1]. This opens the door to applying RL methods, like the use of hierarchical agents or cooperative agents, to this field. Specifically, the idea of using a self-playing agent, a method used in training complex game-playing agents, to create procedurally-generated content is intriguing. We propose a self-playing PCGRL model for level generation, and test it alongside cooperative agents to compare performance.

As an extension of the previous PCGRL work, we evaluate multi-agent reinforcement learning on simple, classic 2D game levels—dungeon levels resembling those from the adventure game ”The Legend of Zelda” and a simple level-generation test of making the longest possible path on a map. These levels have certain elements that can easily be used to judge the quality of the levels produced by PCG. Zelda levels have doors and keys needed to open them while the path-length test is exceedingly easy to measure. Introducing self-play could allow the agent to learn more complex generation behaviors and create content for many more games.

II. Background

PCG is a well-studied topic in game design, and the application of AI techniques has shown promising results. Earlier algorithms treated PCG as a search problem, looking through the space of possible levels and selecting them based on some quality evaluation function [2]. These algorithms typically use some form of evolutionary computing to search through content. Since then, researchers have turned to more complex algorithms to generate content. Dahlskog et al. use n-gram based generators to create ”Super Mario Bros.” levels [3], treating levels as a series of vertical slices and predicting which slice comes next based on the ones before it. Summerville and Mateas turned to LSTMs to parse Mario levels as strings and generate content from that [4]. Others have adapted existing content generation from other fields, like musical composition, to bring inventive methods to game level generation [5]. All of these methods search through the space of possible levels in order to generate content.

Kerssemakers et al. took a higher-level approach and attempted to generate a procedural level generator that met certain constraints from a designer [6]. Since the generator is not fixed, a user could tune the parameters of the generator and produce different outputs based on the content needed. One of the potential methods of generating generators was outlined in a paper by Doran and Parberry which used multiple software agents to generate terrain [7]. Waves of agents iterated over terrain, with each agent performing a different task, such as generating coastline or mountains, or smoothing over existing terrain. Khalifa et al. attempted to use reinforcement learning to train a PCG agent that makes levels for simple 2D games [1]. While other search-based methods look through the space of possible content, using reinforcement learning means this agent looks through the space of possible content generation policies. To our knowledge, this was the first paper using reinforcement learning for PCG.

Reinforcement learning lends itself naturally to training game playing agents: the agents learn a policy from the possible actions, and games often have a score which gives a clear reward schema. For more complex scenarios where the correct path to the reward is less obvious, playing a reinforcement learning agent against itself has been demonstrated to elicit complex behaviors the agent couldn’t learn on its own. As mentioned above, previous work has also demonstrated the benefit of using cooperative or competitive multi-agent scenarios to accomplish different tasks. OpenAI used self-play to train agents to do various complex actions, like push each other out of a sumo ring [8]. Self-play has also been used with great success to play complex games like Chess, Shogi, Go, and Backgammon [9] [10]. Since reinforcement learning has been shown effective in generating content, and self-play has extended other RL methods to more complex scenarios, we chose to leverage self-play to train a content generator to create levels.

III. Problem

A possible solution to our problem definition was designed using an OpenAI Gym interface, a toolkit popularly used for developing and comparing reinforcement learning algorithms. This meant that it would be much easier to implement our representations than by using any other framework. For the purpose of this experiment, we considered two representations, namely Narrow and Turtle. We chose not to pursue Wide as of this time as Narrow and Turtle seemed much simpler to work on as proof of concept. Each representation worked by progressing towards an end goal via the use of rewards to elicit desired behavior. We chose Binary and Zelda as two simple games that could be implemented by our framework easily.

- Binary: A very simple topological game wherein the aim is to modify a 2D map composed of empty and solid spaces, the end goal being that the longest shortest path between any two points in the map is increased by atleast a certain number of tiles.
- Zelda: A simple action adventure game, ported from the popular series of dungeon system games called The Legend of Zelda (Nintendo, 1986). The end goal of the game is to maneuver and guide the titular main character to attain a key and then reach a door, all the while trying not to get killed by an assortment of enemies.

We implement and extend two different representations of game levels as reinforcement learning problems, as defined below.

- Narrow: This representation is the simplest way of representing the problem. It consists of each agent observing the current state and a given location at each step and deciding what change to make at its particular location.
- Turtle: This representation is more complex than Narrow, consisting of the agents being able to move around and modify certain tiles in the vicinity of its path.

The difference between both classes of representations is that narrow representations can only function on a certain number of predefined locations whereas turtle representation can control only variables that are in the vicinity of the current location. For training, we wrote our own Pytorch implementation of PPO. Our implementation consists of four main classes, namely Game (a wrapper for the gym environment), Model (a Neural Network model which is used to define the policy), Trainer (updates the policy), and Math (which is responsible for the main running of the logic). The observation is a tensor of size (84, 84, 4), composed of four frames.

IV. PCGRL Frramework

A. Base PCGRL Framework

The toolkit we used for developing and comparing reinforcement learning algorithms is OpenAI Gym. It works with Python and offers an easy to use interface with access to an environment that enables implementation of any reinforcement algorithm. OpenAI gym also contains a framework for simple games like Zelda and Sokoban. The reinforcement algorithm of choice is Proximal Policy Optimization (PPO), an algorithm which is both more efficient and powerful than many existing algorithms. [11] Its ease of use and high performance makes it fundamental to implementing PCGRL with self-play.

Policy gradient methods are increasingly being implemented in various AI domains. However, policy gradient methods are extremely reliant on how big or small the step size is, resulting in either very slow progress or a huge drop in performance and attenuation due to noise. PPO avoids this by collecting a batch of experiences and splitting it into several minibatches which it then uses to influence its policy decision making. The multiple network updates per training step allow the algorithm to avoid the pitfalls of other policy gradient methods. Once the policy is updated, the previous experiences are discarded and a new batch is collected. PPO is a powerful algorithm that efficiently minimizes the cost function while maintaining low deviation from its previous policy.

The PCGRL pipeline as laid out by Khalifa et al. is a very robust methodology that works well in generating content using reinforcement learning [1]. A random map is generated, and at each time step, an agent takes the map state as observation, determines an action to take, and updates the map, receiving a reward for the update it makes. Their method uses three different generating agents - narrow, turtle, and wide. At each step, narrow agents pick a random tile and can only modify that tile. Turtle agents move through the map - at each step they can either modify the tile they are on or move to a neighboring tile. Wide agents can view the whole game board and change any tile. We implemented the narrow and turtle agents as their network architecture is significantly simpler and the wide agent needs an entirely separate network.

B. Multi-agent Configuration

We test two main configurations with small modifications to the agents' strategies. The first is multiple agents working in tandem to build a map. Each step of the environment, the agents each take one action. The agents are associated with different individual networks, but the networks have the same architecture. Technically, the agents were neither cooperative nor competitive since their rewards were entirely individual. However, we will refer to these agents as cooperative since they are both trying to build a good map, rather than competing with the other agent to get a better reward. To try out different methods of cooperation, we tested two modifications: map restriction, and switching the active agent on negative reward. For map restriction, each agent could only modify half of the map. For negative reward switching, only one agent would act at a time, and that agent would keep working until it received a negative reward or until it had operated for ten steps without getting any reward. A graphic of the architecture can be seen below.

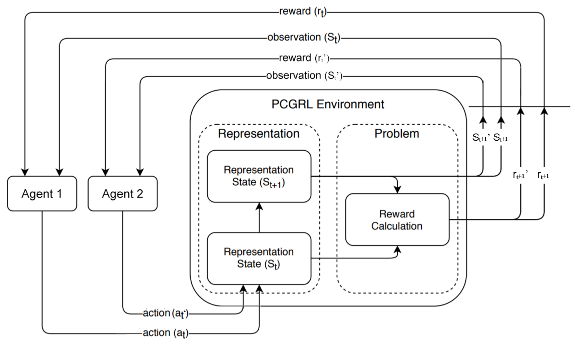


Figure 1: The system architecture for the PCGRL environment for content generation

To train these agents with self-play, two agents operating with the same network modify the same map. Like the cooperative agents, the self-play agents went one after another. At each update of the network, the agent which performed better had its rewards increased by a factor which increased over the period of training and maxed out at double the reward. The agent which performed worse had its rewards decreased by the same amount. Then both rewards were used to update the network. We tested the self-play agents with the same restrictions as the cooperative agents, but abandoned map restrictions for self-play as it proved detrimental to an agent's performance early in testing.

C. Network Architecture and Training Framework

We used pytorch to construct and train our models. The network was constructed with a three layer convolutional network and a final fully connected layer for output, using a rectified linear activation function. At each step, the network took the observation space (current game map) and an agent's position as input, and output an action for the agent to take. The reward was calculated for each agent separately. Our PPO function was an extension of the implementation done by Varuna Jaysiri [12] we modified to handle multiple agents.

During training, each agents' policy was updated every 128 timesteps. PPO operates with batch updates to the network, so after each sampling period of 128 steps, the results were split up randomly into several batches, and for each batch the loss was calculated and used to update the networks. The agents were trained for 10000 updates, with 128 timesteps per update and 2 actions per timestep, for a total of roughly 2.5 million actions on the map. When using negative reward switching, the number of updates was increased by a factor of 1.5 since both agents did not act at every timestep. We trained on as many different representations and restrictions as time allowed.

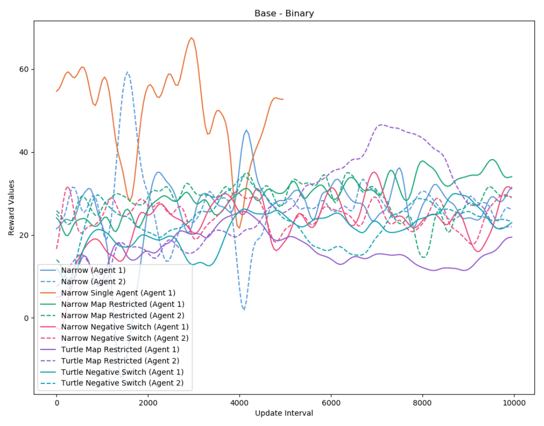
V. Experiments

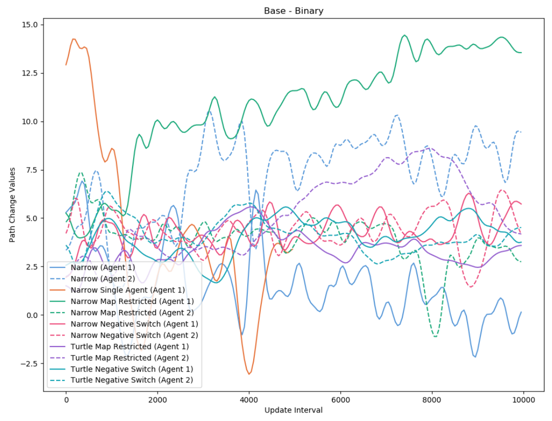
We tested on two game environments, referred to as binary and Zelda. The binary game is exceedingly simple, the agent simply tries to generate the longest possible path on an empty map. The tiles of the map can be either empty space, or a wall. The Zelda environment is more complex. The goal is to have one player character, one door the player needs to get to, one key to open the door. and enough enemies to add difficulty, but not so many that it becomes impossible. Thus the agent can choose from six tile types: empty, wall, player, door, key, and a few different enemies. The agent ran until some completion conditions were met, or a maximum amount of tiles were changed. As discussed in Khalifa [1], restricting the percentage of tiles the agent was allowed to change was key to keeping the agent from remaking the whole map, so the max amount of changes was limited in training and varied throughout testing to optimize the results.

VI. Results

1) Overview: We trained Zelda and binary agents for cooperation and self-play. After training, we got both the reward and path improvement for binary agents, and we got reward for Zelda agents. The figures of reward or path improvement vs. number of updates are shown below. Most agents got 10000 runs, while a few agents only got 5000 runs due to limited training time. We were not able to train all possible combinations of agents and restrictions, but tried to focus on those that we thought promising.

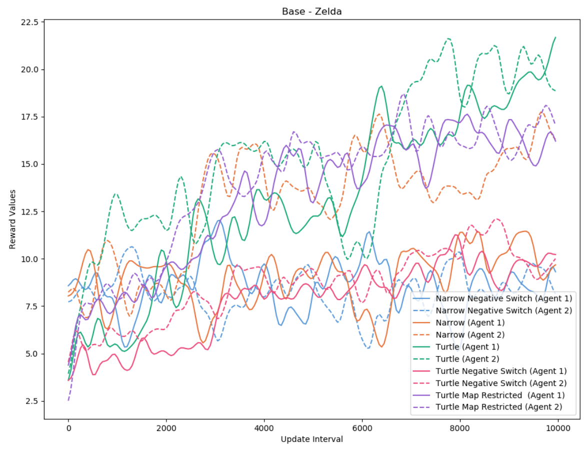
2) Cooperative Binary Agents:





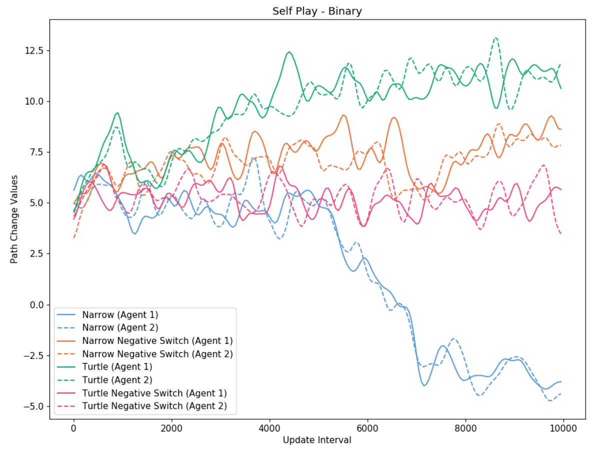
The above graphs show the reward and path improvement of each agent in the cooperative binary runs. Notably, the reward for the vanilla narrow agent is much higher than the rest of the agents, but its path change is significantly lower. This implies that the two agents are not cooperating well. They might each make individual changes that get them a good reward, but the end result is a map that does not function very well since they don't make changes that help the other agent. Unfortunately, their training only ran for half the time as other agents. The other agents all performed about the same, with significant variability in their rewards received from update to update. However, the path improvement for the turtle agents with map restrictions showed significant progress as the updates continued. Despite the fact that the reward oscillated around the same value, the agents cooperated well together and successfully increased the path length. Oddly enough, one agent made most of the positive changes to the map while the other made few meaningful changes. We think this is because the map restrictions cause the turtle agent to focus on one area and prevent them from competing with each other. The other agents showed no significant difference in path improvement as well. The single narrow agent performed poorly, about as well as the double narrow agents.

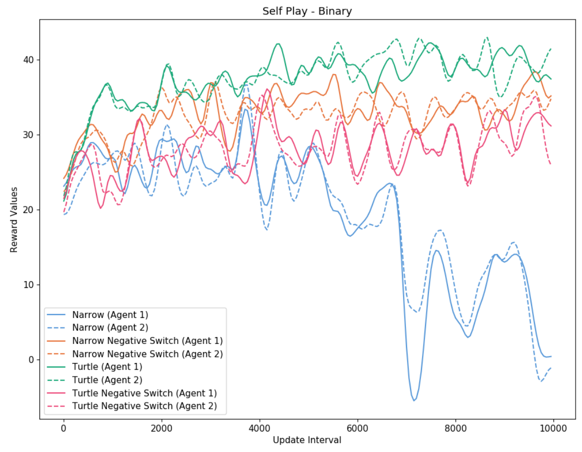
3) Cooperative Zelda agents:



The above graph shows the reward values for the Zelda cooperative agents. The doors, keys, and players on the map generally converged towards one as training progressed. The map restricted turtle agents out performed all the other agents by a significant margin. Similar to the binary problem, we believe this is because map restrictions force the turtle agent to focus and prevent competition. Surprisingly, the negative switch agents actually performed worse than the unrestricted agents, but it is impossible to tell if this result is significant since we weren't able to train the unrestricted agents as long as the others.

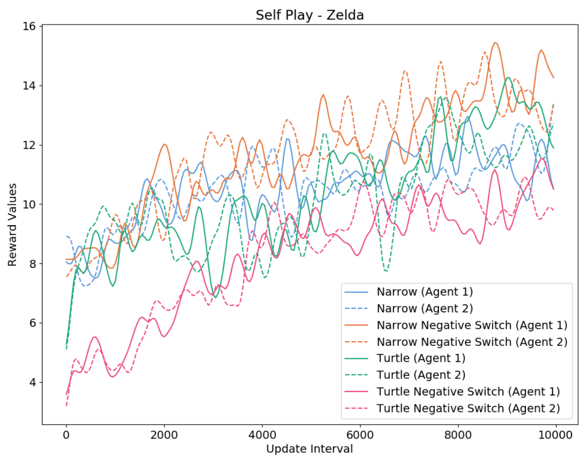
4) Self-play Binary Agents:





For self-play, the binary turtle agent performed much better than any of the other agents. We believe this is because self-play helps the turtle agent to learn better movement around the map. Since the agent that performed better had its rewards increased, the turtle agent that moved to more important locations would have a larger impact on the network update at each step. Agents with negative switch did not perform as well as the vanilla turtle agent. This may be because a negative reward action may sometimes help to extend the path length in the end, and the negative switch further disincentivizes actions that give a negative reward. The agent with a larger sum of rewards at each update updated the map more significantly, so without negative switch, negative reward actions that led to a better path improvement in the end would be incentivized. Curiously, the narrow agent performed very badly. We're not sure as to the cause, and since the other self-play agents did not exhibit the same downward reward spiral, it doesn't seem to be an issue with the update function.

5) Self-play Zelda Agents:



There was no big variation in the way different agents performed. The best performing self-play agent for Zelda was the narrow agent with negative reward switching. Since the negative reward switching forces the agent to avoid spending too long without increasing the reward, the narrow agent may have been more economical with its actions, and made good choices at each location it was assigned. Without any modifications to their behavior, the narrow and turtle agents performed about the same. The turtle agent with negative switch performed slightly worse than all the other agents, but this could just be because the model was initialized with less favorable weights, and training did not progress long enough for the models to stabilize.

VII. Discussion

In general for cooperative agents, the turtle agents outperformed the narrow agents. This makes some sense, as the turtle agents have a choice over where they can modify on the map, so they are better able to target different parts of the map. Because the narrow agents are restricted to a randomly selected tile, they are less able to cooperate effectively. For both the Zelda and binary problem, turtle agents with map restrictions outperformed all other agents. We believe map restrictions are beneficial for turtle agents because they force the the agents to focus on one section of the map and prevent them from moving around too much. Notably, the vanilla narrow agent performed very poorly on the binary problem, actually decreasing the path length for a period of time, and performing worse as training progressed.

For self-play agents, turtle agents once again performed better, but by a smaller margin. for binary self-play agents, the negative switch restriction helped both agents perform better. This restriction is particularly useful for binary self-play agents because it teaches the agent to avoid getting a negative reward. This means the agent is constantly improving the path as much as possible. This compounds with the self-play update method which gives priority to the agent that performed better each batch, making the agent seek out a positive reward. Unfortunately, the vanilla narrow agent performed terribly, but we're unsure if this is due to a fluke in training or something specific to that agent. In Zelda, all self-play agents showed significant improvement over time, with negative switch narrow agents performing slightly better. Without longer training and multiple runs, it is impossible to tell if this result is significant or just specific to these runs.

VIII. Future Work

The most obvious future work is to train these agents for longer and get multiple training instances to figure out which agents actually make good maps. As is, most maps created are mediocre because the agents have not trained long enough. Another important next step would be to explore truly cooperative agents. By giving each agent a portion of the reward that the other agent gets, we could encourage the agents to make changes that would be beneficial to the whole map, and not just the individual agent. It would also be good to test out different network update schemes for self-play agents. Right now, the rewards are increased by a flat proportion for the agent which performed better, but many other schemes could be used to update the network. Finally, we could mix and match different restrictions for different agents, or introduce new restrictions. Trying negative switch and map restrictions shows promise as it might prevent an agent from modifying its part of the map once it is hard to improve. There are many other agent restrictions to try as well.

IX. Conclusion

Without longer training runs and more tests to account for variance, we cannot conclusively say that self-play agents do better or worse than singular map building agents. However, our preliminary results have identified certain restrictions and agent strategies that show promise. For instance, turtle agents with map restrictions performed well for both cooperative and self-play binary agents, and negative switch improved self-play binary agents significantly. Map restricted agents showed strong performance for the Zelda problem. with the right restrictions, self-play has also been demonstrated to increase the reward and path improvement of many agents. We believe that using self-play reinforcement learning to train map building agents is an avenue worthy of further exploration. Our source code is available at https://github.com/Camq543/self-play-pcgrl.

References

[1] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," *arXiv:2001.09212*, 2020.
[2] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, "Search-based procedural content generation," 04 2010, pp. 141–150.

[3] S. Dahlskog, J. Togelius, and M. Nelson, “Linear levels through n-grams,” 11 2014.

[4] A. Summerville and M. Mateas, “Super mario as a string: Platformer level generation via lstms,” 03 2016.

[5] A. K. Hoover, J. Togelius, and G. Yannakakis, “Composing video game levels with music metaphors through functional scaffolding,” 2015.

[6] M. Kerssemakers, J. Tuxen, J. Togelius, and G. N. Yannakakis, “A procedural procedural level generator generator,” *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 335–341, 2012.

[7] J. Doran and I. Parberry, “Controlled procedural terrain generation using software agents,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, 2010.

[8] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, “Emergent complexity via multi-agent competition,” 10 2017.

[9] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play,” *Science*, vol. 362, pp. 1140–1144, 12 2018.

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, 2016.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.

[12] V. Jayasiri, “Proximal policy optimization algorithms - ppo in pytorch,” https://blog.varunajayasiri.com/ml/ppo$_{p}ytorch.html$, 2019.