5 Generators

Four generators were implemented for this project. Two random generators, which have all actions chosen randomly. The first random generator is a Puzzle-Up generator similar to the one already used in Longcat. The other is a Solution-Down generator using a generator method we have designed. The other two generators are based on the PCGRL method. The first is a Puzzle-Up method based on the turtle method from the original PCGRL article [8]. The last is an SD-PCGRL, which uses the same reinforcment learning environment as the random Solution-Down generator but has its actions chosen by a reinforcement learning model based on the PCGRL method.

To summarize, we are investigating two different comparisons: Solution-Down compared to Puzzle-Up generation environments, and random compared to reinforcement learning action decision. A table illustrating these categories can be seen in Table 5.1.

5.1 Difficulty Distribution

Our goal is to generate levels of a specified difficulty. However, we want to compare performance with random generators, where we have no control over the generation. So, to compare, we will observe the difficulty distributions. This will be done by generating a number of levels, using the predictor to estimate their difficulty, and plotting them in a bar graph within difficulty groups. From these distributions, we can then compare how many levels would need to be generated to get one of a desired difficulty, and also further analyze and compare the generators.

	Solution-Down	Puzzle-Up
Random	RSD	RPU
PCGRL	SD-PCGRL	Turtle PCGRL

Table 5.1: Table showing the different categories of generators and which generators belong to which category.

5.2 Random Generator Implementations

This section will review the implementation and generation process for the two random generators.

Puzzle-Up Random Generator

The first generator implemented and tested was a Puzzle-Up random generator already used in Longcat. Puzzle-Up is based on creating the board first and then checking if it is a solvable puzzle. This generator does this iteratively. The process is described in the following pseudo-code:

1. Create an empty level of the desired size.
2. Select a starting position randomly on the board and place the head there.
3. Place a wall randomly on the board.
4. Check if the level is solvable. If it is, save the level as a generated level.
5. If the maximum number of walls is reached, end generation.
6. Go to 3.

The maximum number of walls is decided by a developer beforehand. In the original implementation used in Longcat, it will place all the walls at once before solving, since the developer generally wants a decided number of walls. 1 to 5 walls are usually used in Longcat since the generation is too slow after that. We decided to place the walls one by one and have a maximum of ten walls for our generation tests to give the generator a best-case scenario. The original implementation also has no check for placing walls in ways that clearly make the level unsolvable. We added this in the form of a check for if the level becomes segregated, and a check for more than one dead end. The generator also usually stops an iteration as soon as it has found a level. We wanted to give it a best-case scenario, so we allowed it to continue the iteration after finding a level, which means placing more walls and testing if the resulting level is solvable.

The generator for Longcat is implemented in C. We re-implemented it in Python to compare it more easily with the other models.

Solution-Down Random Generator

The Solution-Down generator is based on the idea of creating the solution and generating the level from it. So we designed a generation process that is based on the actions the player can make in the game. The process is described in the following pseudo-code:

1. Create an empty level of the desired size.
2. Select a starting position randomly on the board and place the head there.
3. Check what spaces are empty around the head.
4. Randomly select one of the directions that has an empty space.
5. Place a cat body in the current position and move the head in the randomly selected direction.
6. If this move creates a turn, place a wall where the head would have gone without a turn. Don't place the wall if there is a cat body there.

7. If there are empty spaces around the head. Go to 3.
8. Create a copy of the level. In the copy, fill all empty spaces with walls, remove all cat body spaces, and move the head back to the starting position. Save this as a complete generated level.

Conceptually, this works from the understanding that if there is a turn in the solution, there must be a wall that the cat can stop at. There can also be a cat body, but since the generation has the same chronology as solving the level, we will also have a cat body there during the generation. So we can generate random actions and take into account the walls that have to be there. This method is guaranteed to give a solution and can generate all possible solvable Longcat levels.

5.3 PCRGL Generators

This section will cover the methodology used to train the PCGRL models. PCGRL is a reinforcement learning method, so both use almost the same Deep Q-Network (DQN) described below. They differ in the environment and reward function used. This is described separately for each model.

Tools

The model was developed in Python using PyTorch and NumPy. The training environment was also developed in Python using the same solver and level class as the random generators. The training was GPU accelerated with CUDA and done on a computer with a AMD Ryzen 5 7600X 6-Core Processor, using 16 GB of RAM and a Nvida GeForce RTX 4060 Ti GPU with 16 GB of VRAM.

Architecture

The model is a convolutional deep Q-learning network based on the one described in the original PCGRL paper. As input, it takes a tensor containing a 2D array representing the current level state. The input is padded and oriented so the current position is always in the middle, so the model can understand its current position without additional input. This results in the input being $(n * 2 - 1)^2$ for a level with a side of size n. For example, a level of size 8x8 will have an input of 15x15.

The network architecture was changed and iterated upon during development. Finally, the one seen in Figure 5.1 was used as it gave the best performance. The network consists of two convolutional layers, a max-pool layer, two more convolutional layers, another max-pool layer, and two fully connected layers that go to the output. The output and the dense layers vary between the two different models as they have different number of actions. The Turtle Puzzle-Up model has the same set-up of actions as the one from the original article[8], four directions and one for each type of block, while the SD-PCGRL has only four actions in total.

MSE was used as the loss function, and the Adam optimizer was used for the backpropagation.

In the training, we used a greedy-epsilon strategy with linear decay, where epsilon starts at 1.0 and ends at 0.0. We also used a replay buffer with a memory size of 600 actions and a mini-batch size of 64. To reduce oscillation, we also used infrequent weight updates, where a separate network is used to update and then occasionally synced with the one used in training. For this, we used a sync rate of 10 episodes.

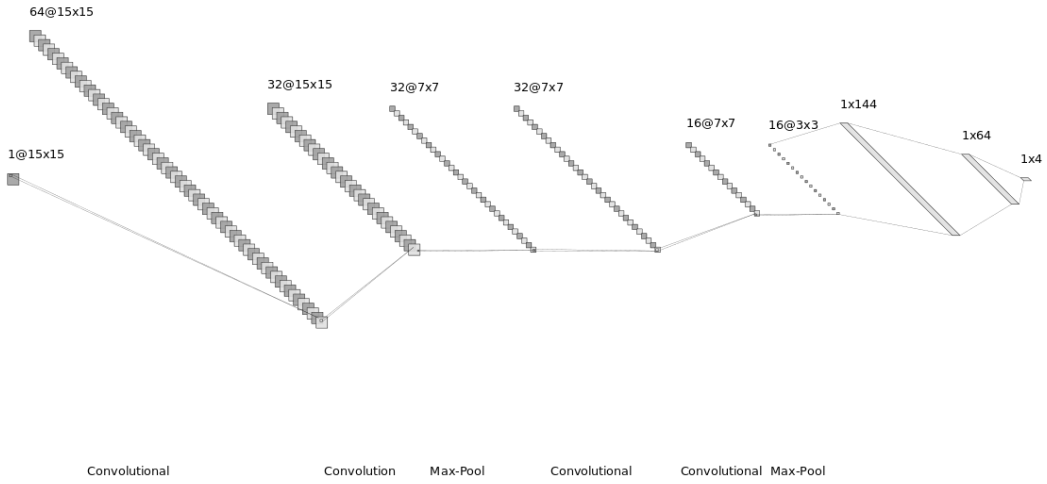

Figure 5.1: Diagram of the convolutional DQN architecture.

Puzzle-Up PCGRL

Environment

This method is mainly based on the turtle method described in the original PCGRL article. The turtle method allows the model to edit one square at a time and move in all four directions to change which square it is editing. So in this environment, it can take seven different actions *up, down, left, right, place empty, place wall, place cat head*. The model has a limited number of squares it can edit and a random initial state of the level. These limitations are important since if the same initial state is always used or if the entire level can be edited, the model can learn one level that gives a high reward and always recreate it. We chose to use an initial state with 19 randomly placed walls and allow the model to edit 25 squares.

The turtle environment is the same as the original in respect to what actions it can perform and what input it gets from the environment. The only difference is that we allow it to edit a larger portion of the level and have a slightly different random setup, as we only have walls and empty spaces placed initially.

Reward function

The reward function here is designed to reward the model for making the level more difficult, but no difficulty can be estimated if the level isn't solvable. Therefore, it also needs some reward for getting the level closer to solvability. The first criteria is the number of heads. The level can only be solved with one cat head, so the model is rewarded for making changes that have the number of heads get closer to one. The second criteria is whether the level is segregated or not. As discussed before, a segregated level is unsolvable, so if the level goes from segregated to unsegregated, the model is rewarded. These things are required for the level to be solvable, but do not guarantee solvability, so the model is also rewarded if the level goes from unsolvable to solvable. Lastly, the model is also rewarded for making the level more difficult, with the reward being proportional to the increase in difficulty as given by the predictor.

This differs from the original PCGRL, which only examined solvability and did not reward steps toward it.

Training

The model was trained for 200,000 episodes, which took around 4 days. Each episode consisted of generating one level followed by one optimization. The level was then tested for that episode. Each test generated eight levels and averaged their difficulty. Other parameters that determine reward were also measured, including the average number of heads, the ratio of segmented levels, and the ratio of solvable levels.

The model was trained with a learning rate of 0.0007 and a discount factor of 0.9. The model never generated any solvable levels during training, but it eventually learned that the board should only have one head around episode 125,000. It did manage to generate non-segmented levels, but not as consistently.

Solution-Down PCGRL

Environment

This method uses a very similar environment to the Solution-Down random generator. The model builds the solution in the same way, but instead of randomly picking directions, the model selects them. So it can take four different actions *up, down, left, right*. If a direction without an empty square is picked, then nothing happens, and the new state is the same as the previous.

This method differs from the original PCGRL[8]. It uses an entirely different environment and set of actions, but it is similar in that it gets the same input from the environment as the original method. It is also similar in that they both describe the generation as a Markov decision process.

Reward function

For this method, solvability is guaranteed, so no reward is needed to aid it in achieving this. Instead, the reward function is only focused on difficulty, rewarding it for increasing difficulty and punishing it for decreasing it.

This is also different from the original PCGRL[8], as it did not give any qualitative rewards besides solvability.

Training

The model was trained for 200,000 episodes, which took around 4 days. Each episode consisted of generating one level followed by one optimization. The level is then tested for that episode. The test consisted of generating one level from each starting position with no random actions and averaging the difficulty between them.

The model was trained with a learning rate of 0.0007 and a discount factor of 0.9. The final model's average difficulty from the test was 48.756.

5.4 Generator test methodology

Each generator was evaluated by having it attempt to generate 20,000 levels. Failed generations and unsolvable levels are noted as unsolvable. Levels already generated by that generator are noted as duplicates. Lastly, only solvable and unique levels are saved, and their difficulty is evaluated and plotted in a difficulty distribution.

5.5 Generator test results

This section describes the results of the tests for each generator.

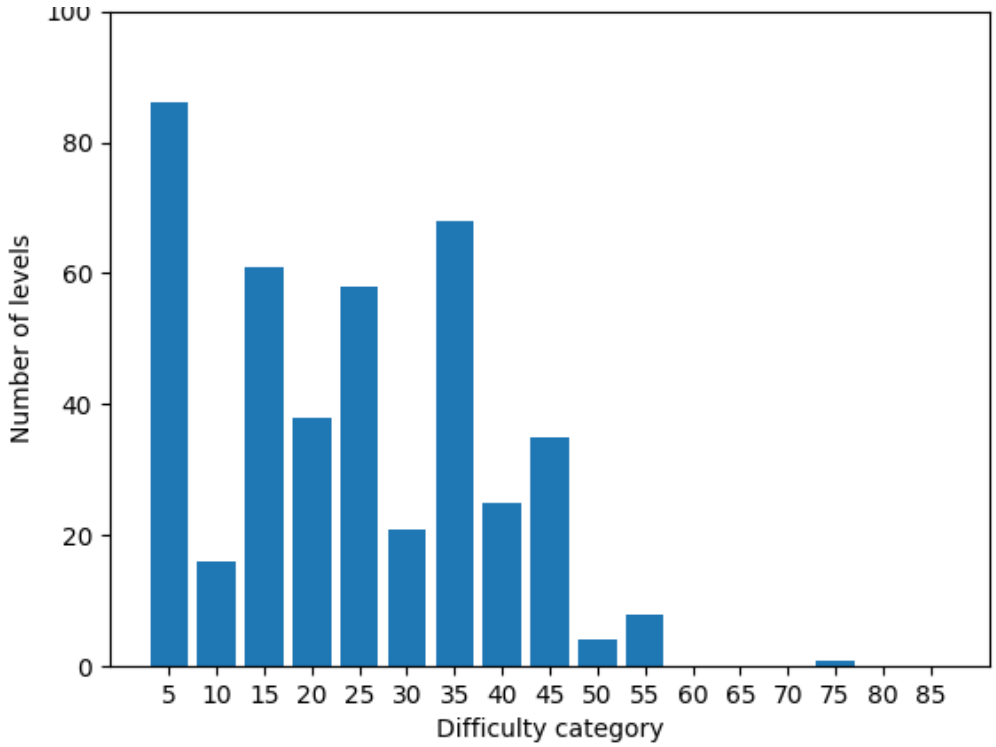

Figure 5.2: Difficulty distribution for unique solvable levels generated by the Puzzle-Up random generator

Puzzle-Up Random Generator

This model differs somewhat from the others in that it runs for iterations, where each iteration may result in no solvable level or several solvable levels being generated. For 20,000 iterations run with the Puzzle-Up random generator, only 1107 iterations resulted in at least one solvable level. Among the generated solvable levels, 759 were noted as duplicates. Only 421 unique and solvable levels were generated. These numbers don't add up to 20,000 since this generator can generate multiple levels during one iteration. The difficulty distribution of the 421 unique and solvable levels can be seen in Figure 5.2.

These levels have very few walls, which gives them a specific appearance. This also has the effect that they feel similar to play. Three levels of varying difficulty sampled from the generated ones can be seen in Figure 5.3 along with their solution graphs.

Solution-Down Random Generator

Since one solvable level is generated for every iteration, exactly 20,000 levels were generated. 1936 levels were noted as duplicates, so 18,064 unique and solvable levels were generated. The difficulty distribution of these levels can be seen in Figure 5.4.

As can be seen from Figure 5.4, the vast majority of generated levels have a low difficulty, but there are still 721 levels over the difficulty 30, which is already more than the total number of unique levels generated from the Puzzle-Up random generator. These levels also have a somewhat distinct appearance, consisting of corridors and small islands of single walls, which can be seen in Figure **??** that shows three levels of varying difficulty sampled from the generated ones, along with their solution graphs.

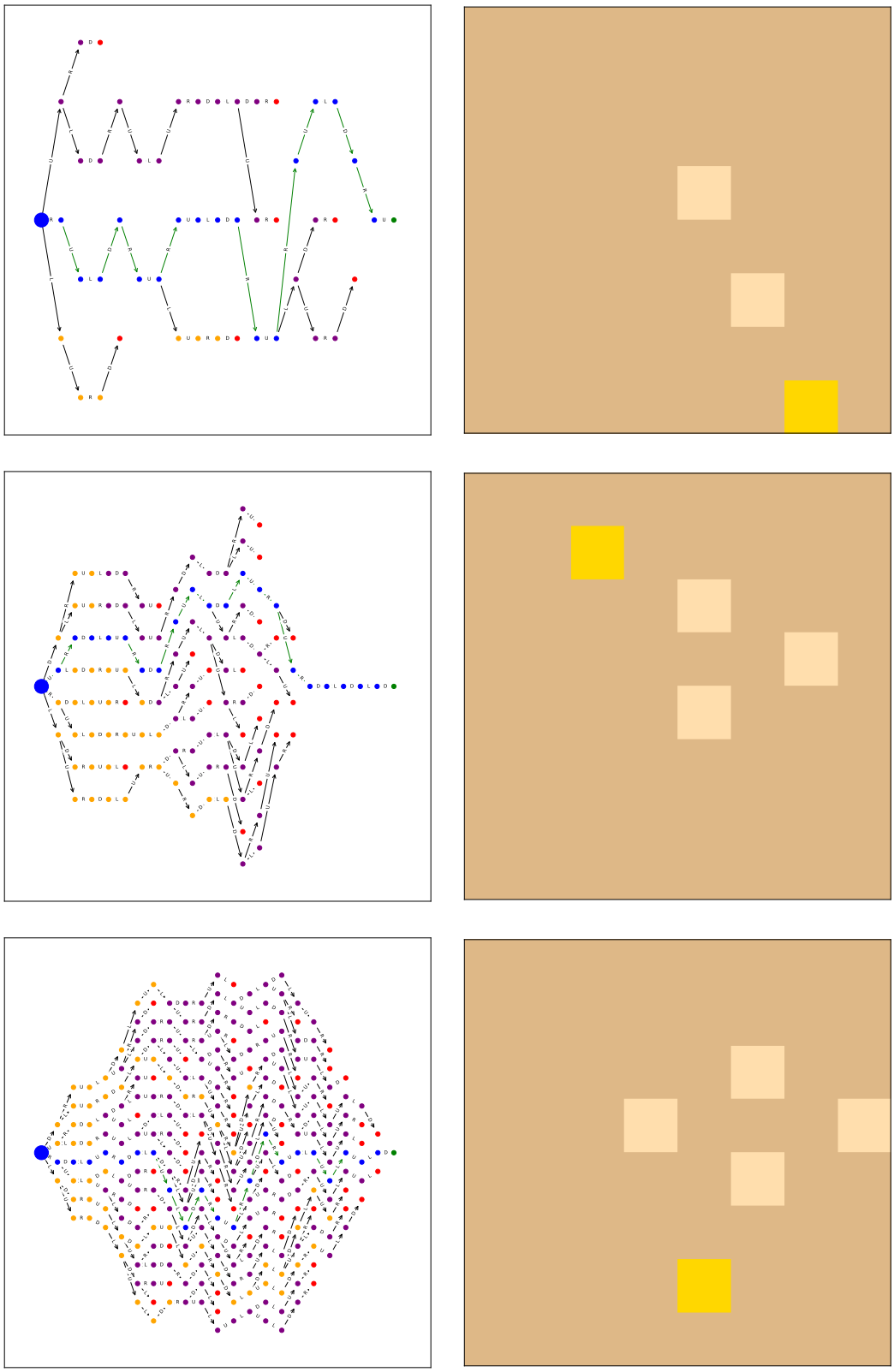

Figure 5.3: Three levels of varying difficulty sampled from the ones generated by the Puzzle-Up random generator.

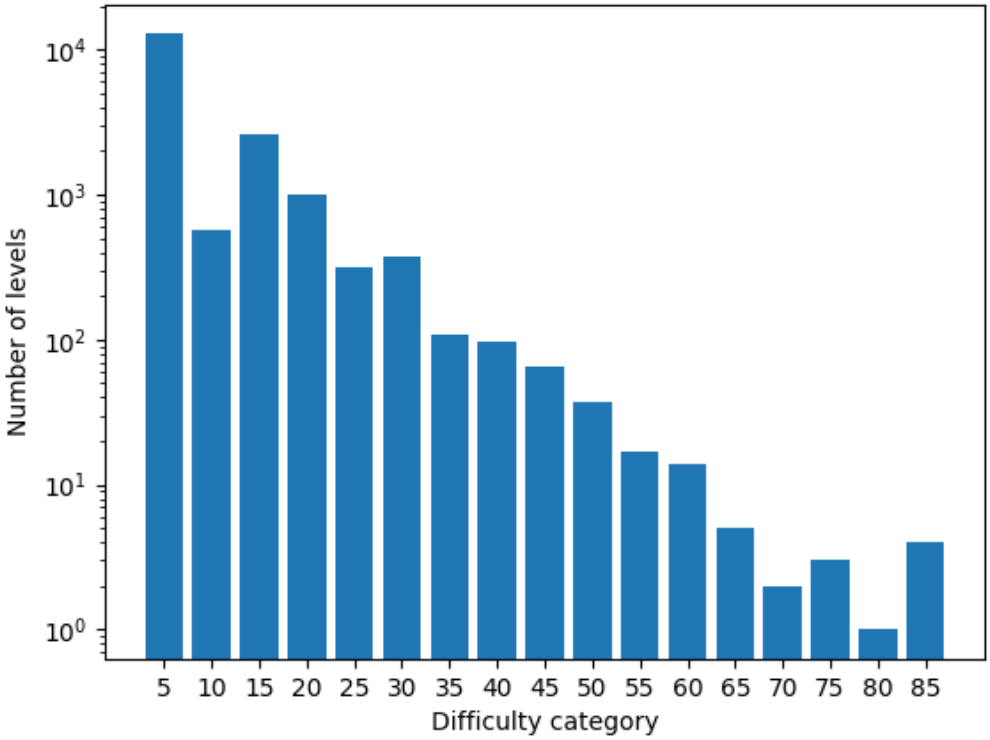

Figure 5.4: Difficulty distribution for unique solvable levels generated by the Solution-Down random generator. The y-axis is logarithmic.

Puzzle-Up PCGRL

The Puzzle-Up PCGRL model did not learn to generate any solvable levels during the training, but we still ran the same tests to see how close it is to finding solvability. Of the 20,000 levels generated, all had one head, 13862 were segmented, but no solvable levels were generated. No level was marked as duplicate. Since no generated levels were solvable, we cannot get a difficulty distribution.

Since all levels had one head and a little less than half were segmented, it seems like the model learned some parts of the reward function, but fell short on learning solvability. There were also no duplicates, which could be promising if it were to learn solvability.

Solution-Down PCGRL

The Solution-Down PCGRL model was tested similarly to others, but since this model is deterministic, only 64 possible levels can be generated when the model is selecting actions. 64 since we get one from each possible starting position. Therefore, we need to introduce some randomness that we call temperature. The temperature gives a likelihood that a completely random action is selected instead of the model's action. So, a temperature of 0.0 means only the optimal action is chosen as decided by the model, and a temperature of 1.0 means only random actions are selected.

To test the model, we had it generate 20,000 levels at eleven different temperatures, 0.0 to 1.0 with increments of 0.1. The number of duplicates for each temperature can be seen in Figure 5.6. As can be seen, the number of duplicates decreases with the temperature. The lower temperatures even have the majority of the generated levels as duplicates.

The difficulty distribution for all unique and solvable levels at each temperature generation can be seen in Figure 5.7. As is expected from the number of duplicates, we can see

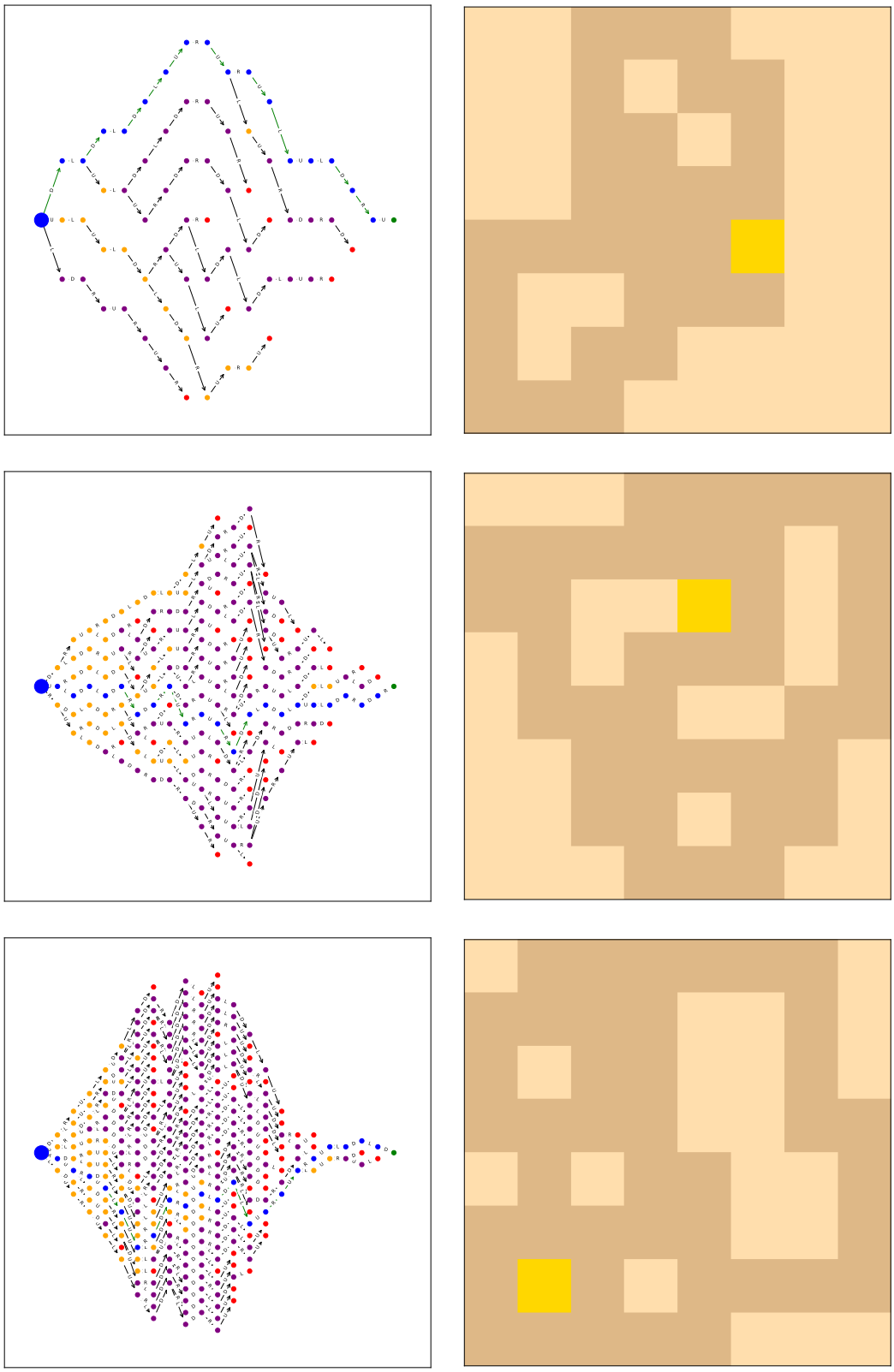

Figure 5.5: Three levels of varying difficulty sampled from the ones generated by the Solution-Down random generator.

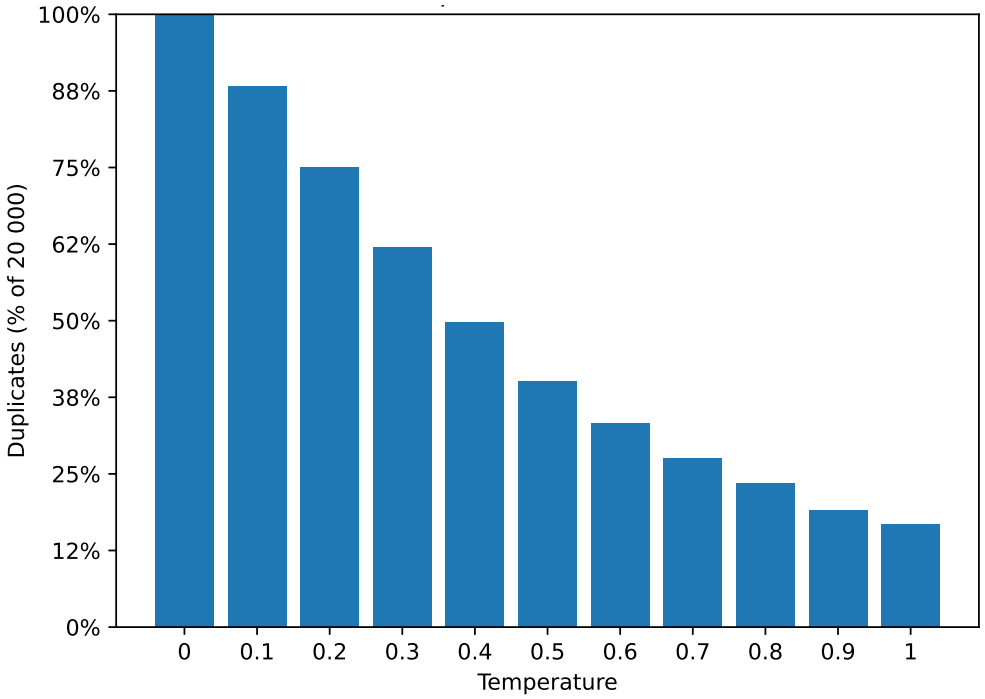

Figure 5.6: Number of duplicates for each temperature of generation by the SD-PCGRL generator

that the total number of levels increases as the temperature rises. However, we can also see that lower temperatures have more levels of higher difficulty. When using temperature 0.0 there are not enough unique levels to create a meaningful difficulty curve. To visualize how well each temperature meets this requirement, we have visualized how many levels within specific difficulty spans each temperature has generated in Figure 5.8. Here, we can see that the completely random generation has the most difficulty for the span of 0 to 20, which can be seen as trivial and easy. However, as the difficulty increases, temperatures ranging from 0.2 to 0.3 have generated a greater number of levels. This is despite them having generated fewer unique levels, as we saw from the duplicates in Figure 5.6. At the highest difficulty span of 80 to 200, the temperature of 0.2 has generated as much as 24 times more levels than the random one.

The levels are visually similar to the ones generated by the random Solution-Down generator. Three randomly selected example levels of varying difficulty generated by the SD-PCGRL generator can be seen in Figure 5.9.

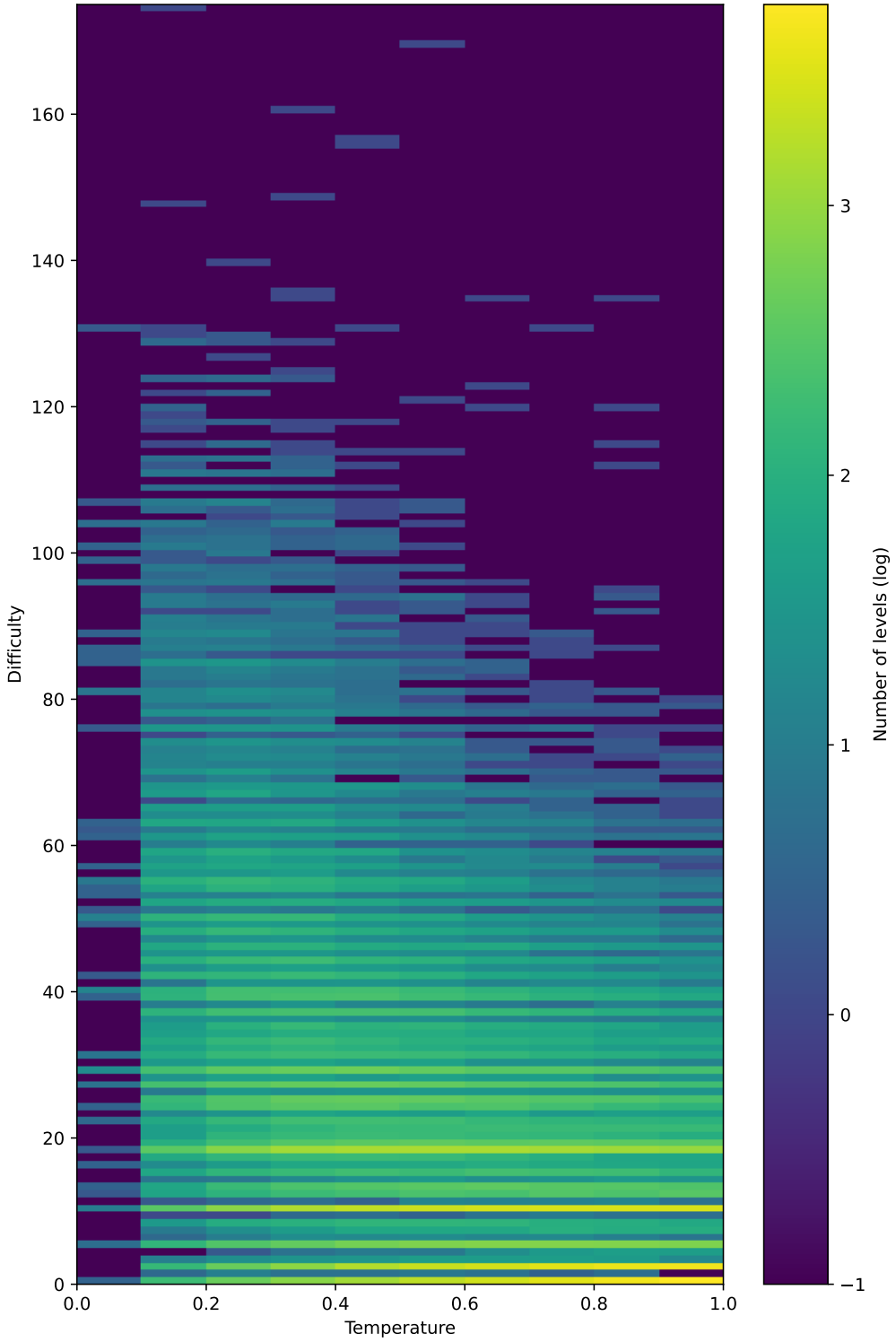

Figure 5.7: Number of duplicates for each temperature of generation by the SD-PCGRL generator

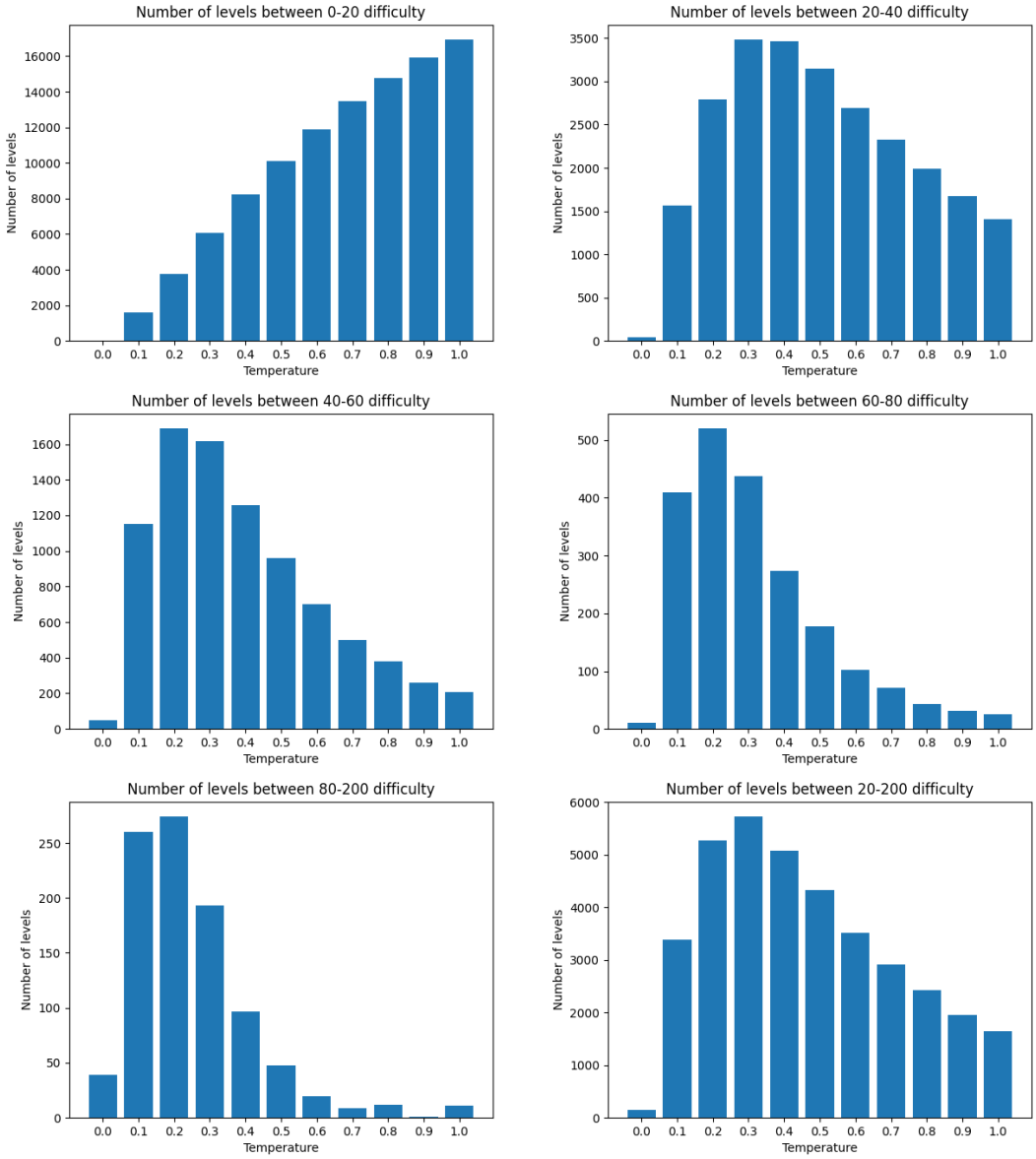

Figure 5.8: Number of levels generated by the SD-PCGRL using all temperatures, for various difficulty spans.

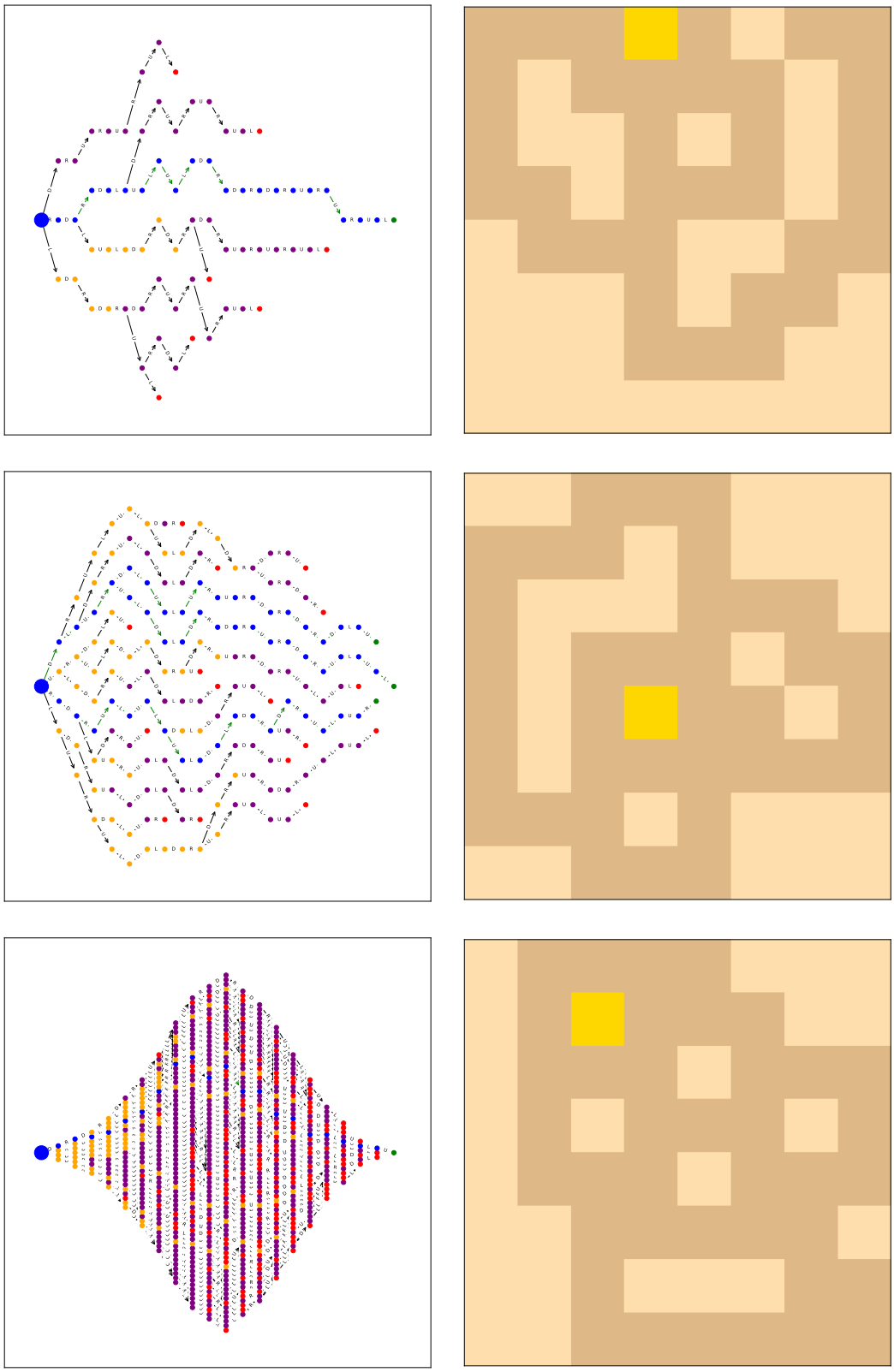

Figure 5.9: Three levels of varying difficulty sampled from the ones generated by the SD-PCGRL generator with temperature 0.3.