Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2025 | LIU-IDA/LITH-EX-A--25/079--SE

Solution-down procedural content generation via reinforcement learning for hyper-casual puzzle games using data-driven difficulty estimation

Lösningsbaserad procedurgenerering via förstärkningsinlärning för hypercasualspel med hjälp av datadriven svårighetsgradsestimering

Elias Johansson

Supervisor : David Bergström
Examiner : Fredrik Heintz

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Procedural content generation via reinforcement learning (PCGRL) has shown promising results in generating solvable levels for puzzle games. Still, little work has been done in exploring its ability to create engaging levels. In this work, we investigate the use of PCGRL in conjunction with difficulty estimation to create engaging levels by making them more challenging. We also propose a novel process for generating levels, called solution-down generation, that guarantees solvability, allowing the model to focus on learning difficulty rather than solvability. To test and compare these methods, we implemented four generators: one that uses random selection and a puzzle-up process, another that also uses random selection but our solution-down process, a third that is based on the original Turtle PCGRL method, and one that uses our proposed solution-down PCGRL method (SD-PCGRL). All the generators were developed to generate levels for the hyper-casual puzzle game Longcat, created by Fancade. In this game, the player attempts to fill different boxes with an elongating cat. Its hyper-casual nature allowed it to have a manageable number of states and actions, and its popularity provided us with a substantial amount of data. We first implemented a difficulty predictor to evaluate and train the models for generating engaging and challenging puzzles. This predictor is based on a previous method for estimating difficulty in casual puzzle games and was trained on player data collected from the Longcat application. We conducted an analysis phase before developing the generator to use this data properly. After the generators were developed and/or trained, we tested them by having each generate 20,000 levels. We then compared the solvability, number of duplicates, and difficulty of these levels. We found that both puzzle-up approaches struggled significantly with generating solvable levels, with the Turtle PCGRL generator unable to generate any solvable levels. We found that the solution-down random generator had fewer duplicates than the SD-PCGRL method; however, the vast majority of these levels were of trivial difficulty. In conclusion, the SD-PCGRL generator showed the best result in creating solvable, unique, and challenging levels for Longcat.

Acknowledgments

Many thanks to my supervisors, David Bergström and Ivar Kamsvåg, my examiner, Fredrik Heintz, and the wonderful people at Fancade.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

People have long enjoyed puzzles. Crossword puzzles, Sudoku, sliding puzzles, and other brain teasers have engaged and intellectually stimulated people throughout time. We continue this tradition today with hyper-casual puzzle games. Digital games with bite sized levels that can be enjoyed by anyone anywhere. Many of these games offer thousands of levels and provide new levels every day to challenge players. This massive generation of content is no small feat. A designer must create the levels, test their engagement quality and difficulty, and then order them in a way that challenges the player appropriately. Doing this for thousands of level can be a time-consuming process.

A solution to this is procedural content generation (PCG), where algorithms are used to automate the generation process. This has been researched and used in many different types of games, both for aesthetics and game content. Still, puzzle games provide a unique problem as the generated levels must be solvable and appropriately challenging [4]. In most games, solvability can be verified or guaranteed through the generation method, but difficulty is more challenging to estimate. It requires a greater understanding of the game's design and how players interact with the game. At the same time, it is also crucial for generating engaging puzzles.

Recently, content generation for other media has progressed significantly through the use of deep learning methods. With these developments, work into using these methods for PCG has emerged, called procedural content generation via machine learning (PCGML)[14]. However, there are some challenges when applying machine learning to games. In the best case, games might have a few thousand levels for training data, but this is not enough for many methods. Games also require levels to be solvable or playable. For some games, even small changes can make a level impossible to complete. What details determine this can be difficult for a model to learn.

Since PCGML emerged, relatively little work has been done on puzzle games, but among the explored methods, procedural content generation via reinforcement learning (PCGRL) has shown promising results for generating levels for the puzzle game Sokoban [18]. PCGRL [8] describes the generation as a Markov decision process, where iterative actions create a level. This requires no training data, which solves one of the challenges with PCGML.

However, PCGRL still does not guarantee solvability and has little qualitative control for generating engaging levels.

So PCGRL could be improved by using generation processes that guarantee solvability and rewarding it for generating engaging puzzles. Hyper-casual puzzle games could be good candidates for exploring this further. Their limited states and actions, relatively high-level quantity, and large player count provide a reasonably sized problem and possibly sufficient data for solving it. One such game is Longcat, which Fancade developed. It has 250 levels, but thousands of users, meaning it is well suited for exploring this problem.

1.2 Assumptions on Engagement and Difficulty

In this work, we aim to explore the generation of engaging puzzles. To do that, we first need to define engagement. This might seem intuitive but the concept of engagement within games is quite fuzzy and has many definitions [3, 9]. It also encapsulate many parts of design such as usability, aesthetic appeal, novelty, involvement, attention. One key aspect is difficulty [9]. Players lose interest if a game is too easy, and players leave frustrated if a game is too difficult. There is a balancing act in design to challenge the player appropriately. As the player plays the game, they also learn its mechanics, meaning progressively more difficult levels are needed to challenge them. So, to create a series of engaging levels, we need to adjust difficulty progressively. This is called a difficulty curve [1].

To adjust this project's scope, we will focus on this aspect of engagement. We will work under the assumption that if we can control the difficulty of generated levels, we can create a series of engaging levels.

1.3 Aim and approach

In this work, we aim to explore whether the PCGRL approach can be trained with a difficulty predictor to create challenging and engaging levels, and how our novel solution-down generation process compares to the one previously used with PCGRL. We call this method solution-down procedural content generation via reinforcement learning.

To accomplish this aim, we will first implement a level difficulty predictor. This predictor will take a Longcat level and return a numeric value that estimates the level's difficulty. It will be used to reward the generators during training for making challenging levels.

We will then create and compare four generators. Two of them will select actions at random: the first, using a puzzle-up process, and the other, using the solution-down process that the authors designed. These two random generators will be used as a baseline. The other two generators will use PCGRL and be trained with the difficulty predictor to generate challenging levels. The first will also be a type of puzzle-up generator based on the Turtle method previously used with PCGRL, and the second will use the solution-down generation process.

The comparison will be done by generating 20,000 levels with each generator and comparing solvability, duplicates, and difficulty of the generated levels.

1.4 Research questions

This project aims to answer the following:

1. What attributes of Longcat levels are relevant for predicting difficulty?
2. What percentage of Longcat levels generated with PCGRL and Puzzle-Up are solvable?
3. What are the differences in the distribution of difficulty for Longcat levels generated with SD-PCGRL compared to those generated with a random generator?

4. What percentages of Longcat levels generated with SD-PCGRL are duplicates, and how does it compare with a random generator?

5. What percentages of Longcat levels generated with SD-PCGRL have a non-trivial difficulty, and how does it compare with a random generator?

1.5 Contributions

This work builds on the concept of PCGRL created by Khalifa et al.[8]. This works contribution is to explore if these models can learn aspects of level design, such as difficulty, which could be used to make more engaging levels. Additionally, this work proposes a different type of generation method than the one used by Khalifa et al. called Solution-Down generation that could, in some aspects, be more effective. This work also tests both methods on the game Longcat, which had not previously been tested.

1.6 Outline

This sections describes the outlining structure of this report. Chapter 1 is an introduction where the motivation for the work is establish as well as its aim and the research questions it aims to answer. Chapter 2 summaries the theory and previous work that the reader needs to understand the work, and also establish what work it is based on. Chapter 3 describes the analysis phase of this work. Before the work could begin some initial analysis of the game Longcat was needed to understand how levels could be interpreted to find their difficulty. Chapter 4 describes the development of the difficulty predictor used in the rest of the work. Chapter 5 describes the work done around the generators. What generators where tested and how they work and differ from each other. The methodology used to test the generators, and the result of these tests. Chapter 6 discusses the methods and results found in all the three previous chapters. Chapter 7 gives the conclusion of the work and clearly answers the research questions.

2 Theory

This chapter will cover background knowledge that is important for understanding the rest of the report. It will also discuss previous work in procedural generation and machine learning that this work builds upon.

2.1 Background

This section will describe the game Longcat, the game for which the generation methods will be tested. It will then introduce the concept of Puzzle-Up and Solution-Down puzzle generation. Finally, some basic concepts around reinforcement learning and Deep Q-Learning (DQN) will be introduced.

Longcat

Longcat is a casual puzzle game for mobile phones developed by Fancade. Fancade is a small game company from Sweden, most well-known for the game development application by the same name. Longcat started as a smaller game within the Fancade application but has been developed further as a standalone application. In Longcat the player solves puzzle levels. Each level consists of a grid of walls, empty spaces, and one cell containing the cat head. The player can control the cat by swiping in a direction. The cat will then elongate by moving its head in that direction, filling each cell as it goes until it hits a wall or a part of its own body. It will then stop. The player can then move again with the cat's head moving in the new direction, filling spaces as it goes. The player can not move in a direction if it is immediately blocked. The goal is to fill every empty space in the level with the cat. If the player moves in such a way that there are no legal moves, i.e., the cat's head is surrounded by walls or the cat's body, then they have failed and have to restart the level. The player may also restart the level at any time. A screenshot from the game can be seen in Figure 2.1.

Puzzle-Up and Solution-Down puzzle generation

There are many methods for PCG and many ways of categorizing them. One categorization we found very helpful is Puzzle-Up and Solution-Down, as described by Jonah Segree in his presentation at ThinkyCon 2024 [12]. In Puzzle-Up, we start by randomly or semi-randomly

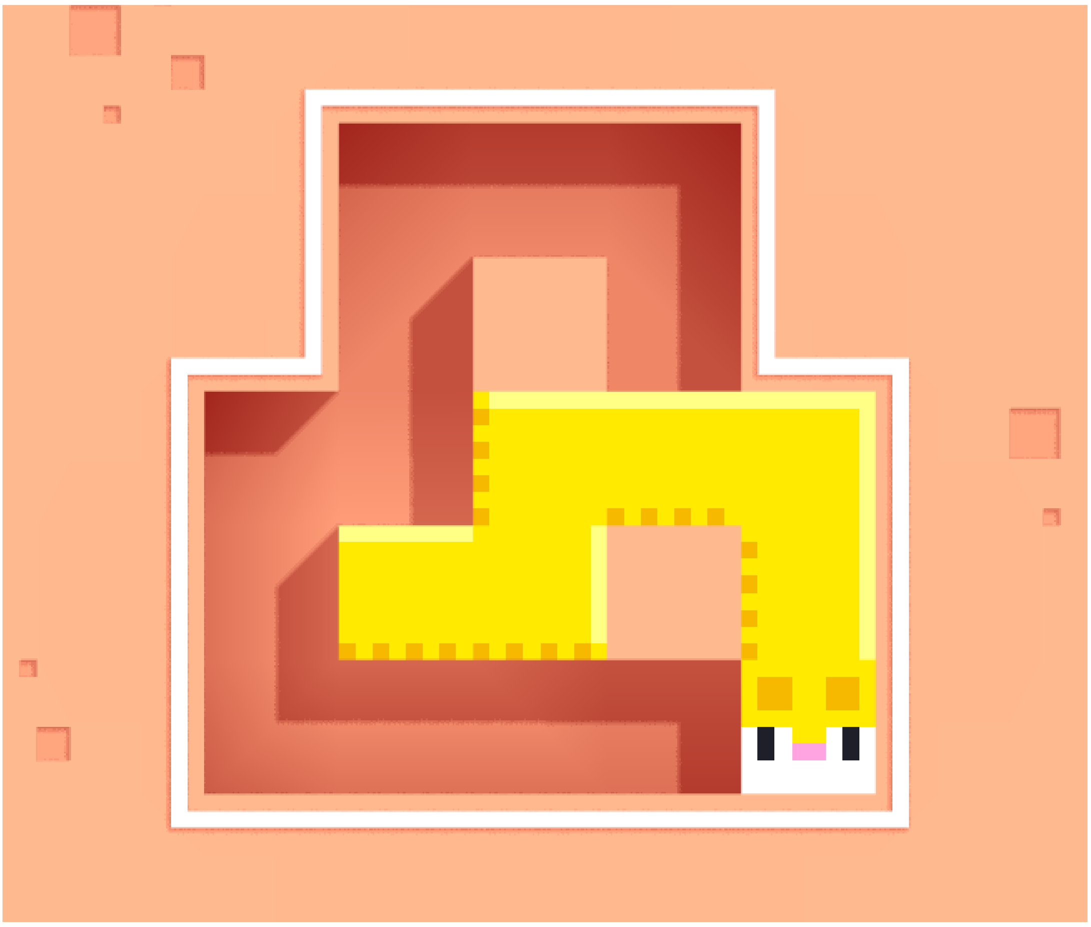

Figure 2.1: Example image from the game Longcat.

building the level's state by placing all necessary parts of the level into an empty level. Then, we find all solutions to the level. There may be no solution, in which case the generation must be reattempted. This generation type is thorough as it can find any possible level, but this can also make it very slow, as it may generate many unsolvable levels. As an example, we will look at a simple maze game. In this game, you can move in four directions but not through walls. The goal of the game is to find the coin. An example of a Puzzle-Up generation for this maze game can be seen in Figure 2.2. We first start with an empty level. We then randomly place one player, one coin, and some walls. Last, we attempt to find a solution by using a search algorithm. In our example, we can see that there were two solutions, but there could also have been no solutions if the walls blocked all paths to the coin, in which case we would have to restart.

In Solution-Down, we first create the solution to the puzzle level. This process must be implemented specifically for each game, as the solution we create depends on the game's possible actions. In contrast, Puzzle-Up allows us to use the same process but with different puzzle elements. We then create the level from that solution, keeping in mind the constraints needed to preserve the solution. This will always create a solvable level and is generally much faster than Puzzle-Up since it does not need to regenerate levels if they are not solvable. However, there may be more solutions to the level that are not considered from just the generation process. Also, this type of generator can have a tendency to produce very trivial puzzles. Depending on the implementation, Solution-Down may also not be able to generate certain levels. i.e., it may only be able to generate a sub-set of all possible levels. As an exam-

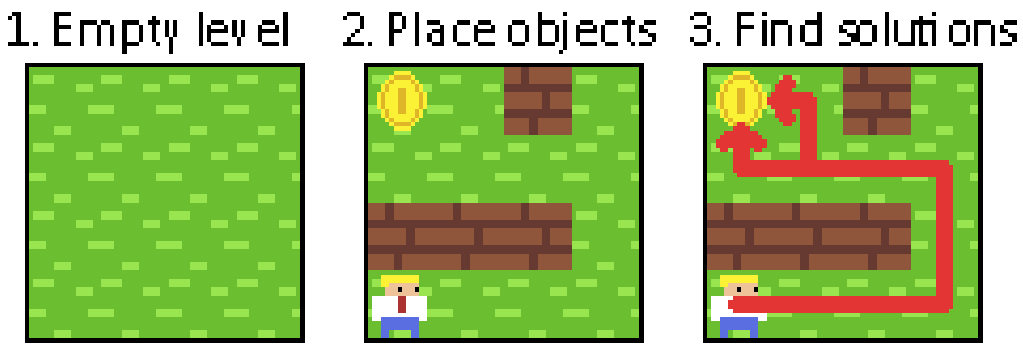

Figure 2.2: Example of a Puzzle-Up generation method for a maze game.

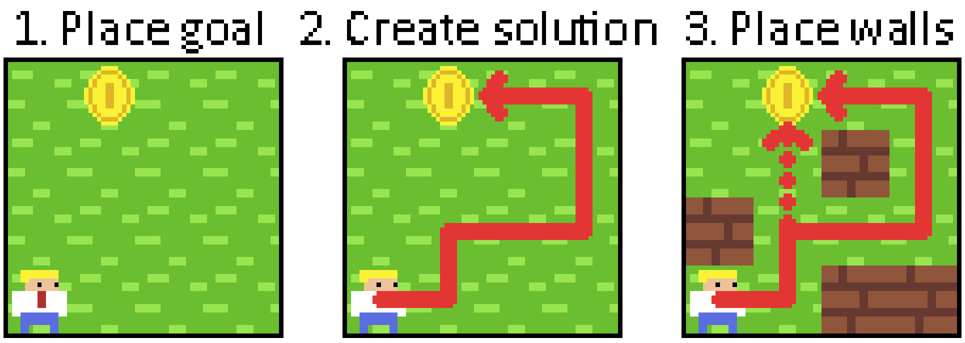

Figure 2.3: Example of a Solution-Down generation method for a maze game.

ple, we will look at a Solution-Down generation method for the same maze game as before, illustrated in Figure 2.3. First, the player and goal are placed randomly. Then a random solution is found. Last, walls are placed randomly around the solution path. Since the walls can not be placed on the solution, we are guaranteed to have a solution to the level. However, as can be seen, an additional solution exists in the level.

Reinforcement Learning

Reinforcement learning is different from its supervised and unsupervised learning counterparts. Instead of looking at structure in data, reinforcement learning learns from interaction with an environment. Formally, in dynamic systems theory, it is defined as the optimal control of an incompletely-known Markov decision process. In simple terms, we have an agent in an environment. It can, in some way, perceive and interact with its environment. It also has some goals connected to its environment and a way of evaluating how well it is reaching its goals. This "goal evaluation" is called a *reward function*. Since the process is an incompletely-known MDP, the agent will not necessarily always know what state an action will lead to. The methods of reinforcement learning then work to find what actions are optimal in each state to maximize the reward function.

Reinforcement learning has some unique challenges with the trade-off between exploration and exploitation. For the model to gain reward, it needs to continue doing actions it has previously found effective, but to find these actions, it first needs to try unexplored actions. The problem is that both need to be done to solve the problem, but it is unclear how much of each is optimal. The general solution is to have a thorough exploration at the beginning of the training so the model can find useful actions and then exploit them later. This strategy is called *epsilon-greedy* policy. the exploration-exploitation dilemma does not seem present in other machine learning methods.

However, reinforcement learning also has advantages in that it considers the whole problem of a goal-oriented agent. Many other methods look at sub-problems and never consider how the model will operate as a whole. Reinforcement learning can also operate in uncertain environments and will generally adapt to this uncertainty.[15]

Deep Q-Network

A common reinforcement learning method is Q-learning, where the agent learns a Q-function that, given a state, returns the expected value of each action in that state. The agent then explores the environment and updates the Q-function with the actual given values of actions. Traditionally, in Q-learning, the function is saved in a table, with each state and action pair, but this is unrealistic for problems with very large or continuous state spaces.

A solution to this has been Deep Q-networks (DQN), which instead use a deep neural network to approximate the Q-function. This works by giving the neural network the state as input and having it return its estimated value of each possible action. During training, when the environment is explored, the actual value of actions is used to train the neural network as usual with back-propagation. This allows us to represent very large state spaces as a few variables, and also allows for the use of continuous values.

There are several techniques that are often employed with DQN. The first is called experience replay. This works by using a replay buffer that stores experiences in the form of states, actions, and the reward given. The network is then trained on random mini-batches of these experiences, rather than just the most recent ones. This allows the same experience to be used in learning multiple times, thereby improving sample efficiency. Additionally, by selecting them randomly, we break the temporal correlation between experiences, leading to more stable learning.

The second is using a target network. This approach utilizes two neural networks. One is the target network and is used to calculate the Q-values during training, but is not directly trained on with the experiences. The other is only trained on, and done so frequently. Then, it is used more infrequently to update the target network. This prevents rapid oscillation in the learning.

2.2 Related work

This section will review related work for this project somewhat chronologically. First, we will look at procedural content generation in general, then the use of machine learning in the field, its application to puzzle games, and PCGRL, the method this project directly builds upon. Lastly, we will review a method for difficulty estimation that is used in the project.

Procedural Content Generation

PCG for puzzle games has been a long-standing problem with a multitude of solutions. This can be clearly seen in the work of De Kegel and Haahr [4], where the generation of everything from *Sokoban* and riddles to mazes and narrative puzzles is explored. One or more games are presented in each category, and each is presented with multiple PCG methods. Some methods even date back as far as 1997, as in the work of Murase, Yoshio, and Matsubara [11].

With these methods, there also arises a need for difficulty estimation. As described by and explored in work by Spierewka, Łukasz, and Szrajber [13], puzzles are designed around putting the player in a flow state where they are adequately challenged in such a way to facilitate engagement with the game. Spierewka et al. use the work of Van Kreveld, Marc, and

Löffler [16] to create an estimator that aided in generating puzzles for the game *inbento*. However, their conclusion also discusses the potential use of machine learning methods within this area.

PCG via Machine Learning

Work on PCGML is explored in the work of Summerville et al. [14]. They survey different works where different machine learning methods are used to generate procedural content for games. The focus is on *functional* generation. In other words, non-cosmetic changes that actually affect the gameplay. They mention generation for platformers such as *Super Mario Bros*. Generation for card games such as *Magic the Gathering*. Generation for top-down action games such as *The Legend Of Zelda* and much more. Among the many examples, there are none for puzzle game generation. The survey also describes several challenges with PCGML for games. Among them is the problem of ensuring that the levels are playable or solvable. As well as the limited amount of data available for training data for games compared to other media.

Liu, Jialin and Snodgrass [10] conducted a similar review of procedural content generation aided by deep learning, motivated by the increased number of papers on the matter in recent years. This review has a broader perspective, also looking into methods used for generating text, character models, textures, audio and other content for games. Even with this breadth of exploration in the area, very little is discussed about puzzle games within the review.

PCGML for puzzle level generation

However, there are examples of puzzle-level generation using machine learning. Of interest is the work of Zakaria et al. [18], where multiple methods are tested for the generation of *Sokoban* levels. The methods tested include generative adversarial networks (GAN), variational autoencoder generative adversarial networks (VAEGAN), variational autoencoders (VAE), long short-term memory sequence generators (LSTM), procedural content generation via reinforcement learning (PCGRL), and generative playing networks (GPN). Even though the paper is an experimental study, it still achieves some promising results. Some methods, such as PCGRL, gave more than 80% playable puzzles, high diversity, and few duplicates.

Procedural content generation via Reinforcement Learning

Procedural content generation via Reinforcement Learning was first proposed in 2020 by Khalifa et al. [8]. The generator they show in this work frames level design as a game, where stepwise actions are taken to create the level. By using this framing, reinforcement learning can be used to learn actions that maximize some wanted level feature. This can be done completely without training data. Instead, the reinforcement model explores the design environment itself to generate levels, and a separate evaluator rewards it when it creates desirable levels.

Their work explores three ways of representing the generation process for discrete 2D levels as a Markov decision process (MDP). Narrow, where the agent is given the current state of the level and a position and can change the square at its position or move on to the next square. Turtle, which works similarly, but instead, the agent may move in one of four directions instead of going through the levels square linearly. Wide, where the agent is given the entire world state and may change any square at any location.

They tested the three methods on three different tasks: binary, where the task is to create a maze of a specific length; *Zelda*, where a dungeon is created and needs a specified number of objects; and Sokoban, where a solvable Sokoban puzzle is created with the same number of boxes and targets. Among these games, Sokoban is the one most similar to Longcat, as they both are puzzle games, but with Sokoban having more complex puzzles and less restrictive movement than Longcat. They found that the method was fast and could generate playable Zelda and Sokoban levels with high accuracy. However, the levels were generally found to be very easy.

Since its introduction, more work has been done to explore PCGRL. A large downside of reinforcement learning in general is that the training time can be very long when compared to other methods. In 2024, Earle et al.[7] attempted to mitigate this by implementing environments in JAX, an environment which allows both learning and simulating to happen in parallel on the GPU. Work on controlling the generation has been done by giving a desired number of level features, as can be seen in the work by Earle et al. in 2021[6]. This controlling has also been explored by giving more abstract instructions and in natural language, as seen in the work by Baek et al. in 2025 [2]. To the best of our knowledge, no work has been done to control the difficulty of the generated levels.

Automated Puzzle Difficulty Estimation

Difficulty is, as discussed previously, an important part of game design. So, tools for estimating difficulty are of use for designers. Kreveld et al.[16] explore the development of such tools in their work. They look specifically at three casual puzzle games: Flow, Lazors, and Move. These are all grid-based casual puzzle games, making them very similar to Longat. They describe a method where they first select some variables from the game, such as level size, solution length, and the number of certain objects in the puzzle. They then conduct a study in which they have users play levels and rate their difficulties. With this, they used linear regression to fit the variables to the user-described difficulty of the levels.

This quite simple method gave very good results for the given games. The method does require some understanding of the game for the variable selection, and more sophisticated methods than linear regression have appeared since its publication. However, it still seems like a good baseline method for difficulty estimation.

3 Analysis

To predict difficulty, we first needed a deeper understanding of how Longcats' is designed. We needed to understand what possible states and actions are actually available to the player and how players parse this information. Therefore, we started with an analysis phase. The method and results of this analysis are described in this chapter. It is split into two sections, one that looks at the levels and what variables can be gained from them, and another that looks at the player data to give a value on how difficult the existing levels are.

3.1 Level Analysis

This section has two parts. The first part describes the graph representation used to analyze Longcat levels. The second part lists and describes the relevant variables we found in analyzing the graphs.

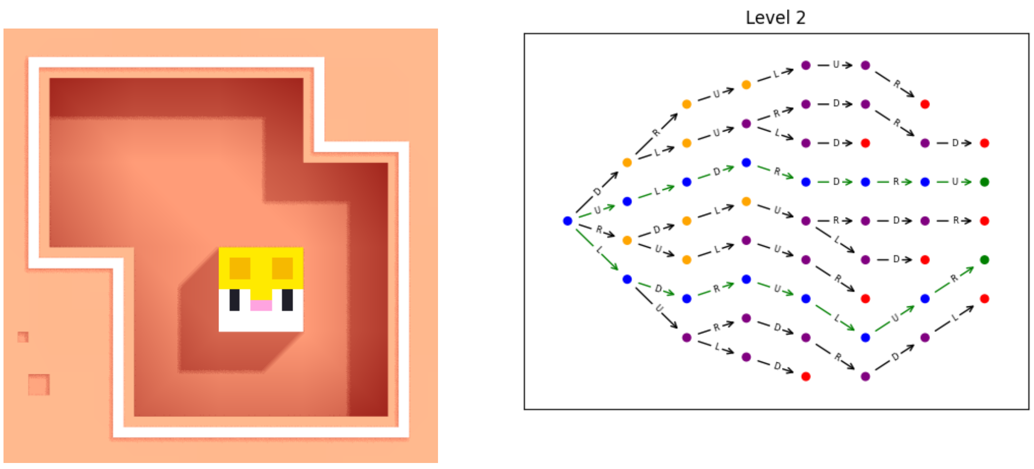

Figure 3.1: Level 2 of Longcat (left) and its solution graph (right)

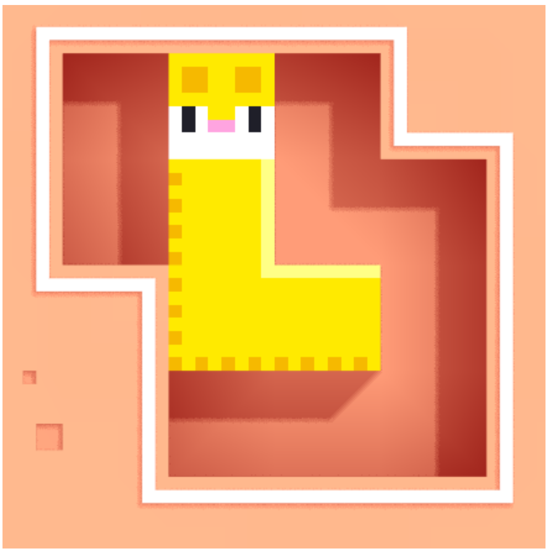

Figure 3.2: Level 2 in a dead state.

Graph Representation

Longcat levels don't always have an apparent difficulty. Unlike games such as the ones analyzed by Kreveld et al. [16], where more puzzle elements and level size correlate with difficulty, Longcat has less clear variables. A Longcat level with a large size can be trivial, and the same level can change substantially in difficulty by adding one or two walls. Because of this, we choose not to look at the levels in the grid representation but instead as a graph.

In this graph representation, each board configuration, or state, is represented by one node, and an edge represents each action. So when the level starts, we are in a start state represented by a node. When the player moves in a direction, it adds cat cells and moves the head so that we are in a new state and, thus, a new node. These two nodes are connected by an edge representing the directional action used. With this, we can use a simple breadth-first search to find all possible states of a specific level and what actions connect them. We can then write this as a graph of nodes and edges. We have chosen to call this the *graph representation*.

As an example, we will look at level 2 in Longcat and its corresponding solution graph, which can be seen in Figure 3.1. The nodes go chronologically from left to right, so the left-most node is the start state. Each edge is labeled with the action it represents. *U* for up, *D* for down, *L* for left and *R* for right.

The graph is also color-coded. Each green node represents a success state, which is a state where the level is completed. Each red node represents a fail state, which is a state where the levels has to be restarted. Blue nodes are states from where a solution is reachable, and edges are green if they connect two blue nodes. This makes it easy to see all solutions in the graph. As we can see in Figure 3.1, level 2 has two solutions.

Purple nodes are *dead states*. A dead state is a state where the level is separated into two areas of empty cells. Figure 3.2 shows an example of this. We choose to track this since it is evident for most players that there is no solution from a dead state, and so it is not a state the player will continue to explore from. This means many dead states will likely not lead to a more difficult level.

Yellow nodes are *indeterminate states*. Indeterminate states are defined by not being solutions, success, failure, or dead states. Meaning that these states will not lead to a solution, but this is not obvious, or at least not as obvious as in a dead state. We call them indeterminate states since it is difficult for the player to determine if they have found a solution from them, not be confused with the mathematical concept of determinants.

Graph Variables

With this graph representation, we can now better understand how many possible ways a level can be explored and how many of these paths lead to a solution. The intuition is that a large and more complex graph, with many branches and few solutions, will have a more difficult level. We analyzed a few different attributes of the graphs to attempt to extract this information.

- **Number of states**. This is the number of nodes in the graph. A greater number of states means more states to search before finding the solution.
- **Size**. The number of cells in the grid representation.
- **The number of branches**. If a node has more than one out degree, we say that node has a branch for each additional out degree above one. In the level this equates to the number of choices the player can make. If every state, aside from the success state, has an out-degree of one, then the level is trivial since the player can only make one move in each state and will then eventually reach the end automatically. Intuitively, the inverse should also be true. A great number of branches should lead to many choices having to be evaluated and a more difficult level.
- **Number of solutions**. The number of success states. A great number of success states means more ways of finding a solution and could make the level easier.
- **Number of fails**. The number of fail states. As with a great number of success states a great number of fails states could lead to a more difficult level.
- **Shortest solution length**. Number of nodes in the shortest path from the start node to the nearest solution node. This equates to the smallest number of actions needed to solve a level.
- **Number of solution branches**. Solution branches are the number of branches from a node that is part of a solution (i.e., blue nodes). This equates to the number of actions the player can do incorrectly when following a solution.
- **Number of non-dead states**. As discussed previously, dead states can create parts of the graph that the player won't explore.
- **Number of indeterminate states**. Unlike dead states, these should give a good indication of how many states the player has to search through to find a solution.
- **Number of indeterminate branches**. Indeterminate branches are the number of branches from an indeterminate state. This equates to the number of choices in the level, where it is unclear whether the level is unsolvable.

Not all these variables were defined at the start of the analysis. Some were added and changed during implementation to adapt to observed problems and player behavior. Not all of them proved useful, which was expected, but exploring all of them and later removing irrelevant ones, as done by Kreveld et al. [16] in their work, seemed like a good method.

Data Analysis

From the level analysis, we can now represent the level's complexity through variables. Now we need to analyze how these variables correlate with the actual difficulty of a level. So, we need a set of levels labeled with their difficulty.

Kreveld et al. [16] achieved this by doing a user study where each participant was asked to rank levels after playing them. We decided to use large-scale player analytics to determine

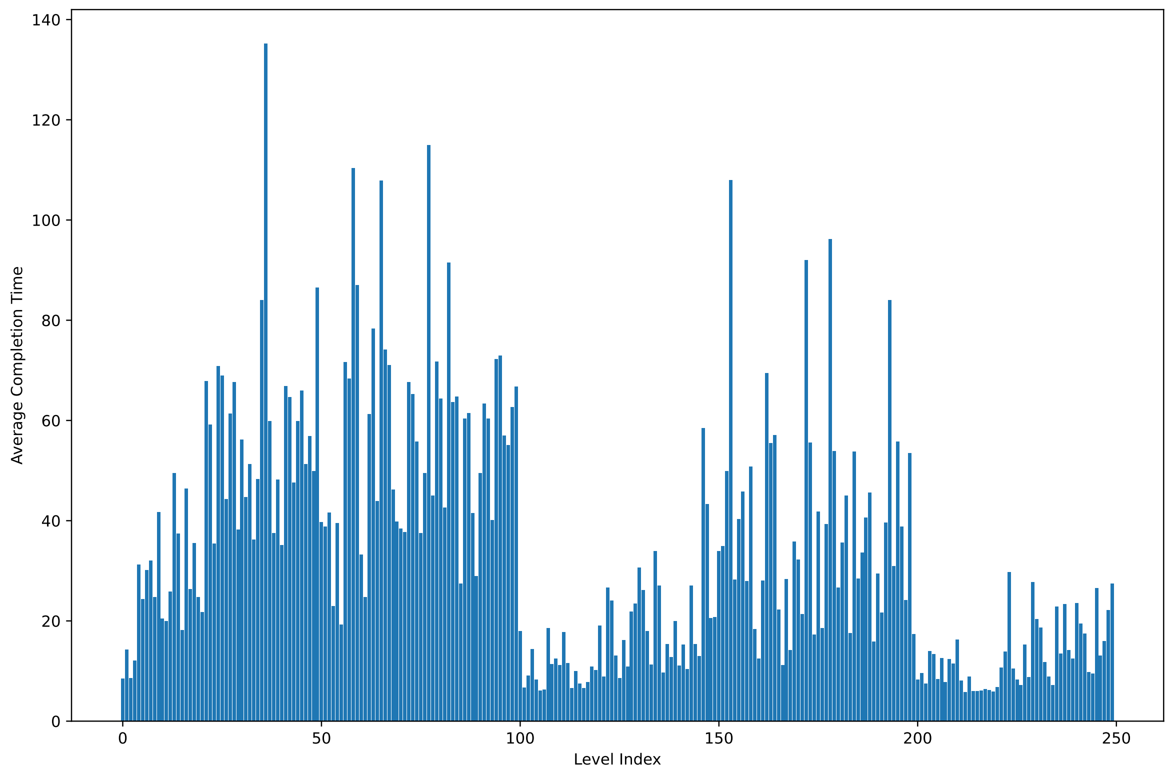

Figure 3.3: Average completion time for all 250 levels in Longcat.

difficulty, so instead, we looked at the player data from the 250 levels that were already part of Longcat. We could extract each player's time to complete a level from these player analytics. With around 900-12000 players per level, we could get a strong average completion time for each level. This number is assumed to correlate with difficulty. This assumption is not completely true. A level can take many time-consuming actions to complete but not be difficult, but we assume that this time is generally outweighed by the time it takes to think about and solve a level for most non-trivial levels.

Figure 3.3 shows the resulting average completion time for all levels. As expected, we can see the completion time rise during the period level 0 to around 20. This is because the first levels are designed to teach the player how to play the game. So, they are designed to rise in difficulty. Interestingly, we can see that levels 100 - 150 and 200 - 250 have a shorter completion time. This is also by design. The first 100 levels of Loncat are designed by hand, with only some generation assistance used for levels 50 - 100. Levels 100 - 250 are essentially completely procedurally generated using a naive approach. This generator was instructed to create more difficult levels for the range 150 - 200 and easier ones for the rest. Another observation of note is level 37, which has a much larger completion time.

We also got a distribution of how many players have played each level from the data. This can be seen in Figure 3.4. As expected, the player count drops off initially as players lose interest. We can also see a player drop-off after level 37, meaning many players started level 37 but never completed it. From these observations, level 37 seems like an outlier, which is

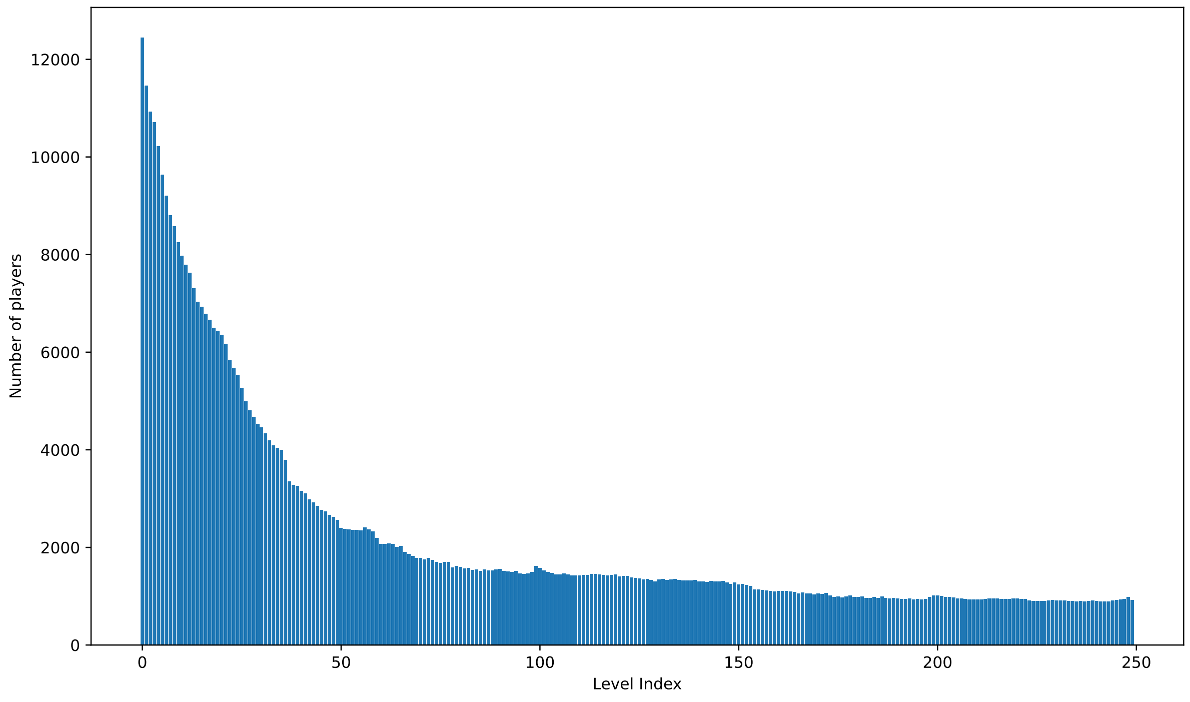

Figure 3.4: Player distribution over all 250 levels in Longcat.

also true in the graph representation seen in Figure 3.5, as it has one of the most significant numbers of nodes in the game.

We then calculated the graphs for all 250 levels, extracted the variables, and plotted them against the average solution time. We also fitted a simple linear model for each variable to observe the corresponding r-values. The results of this can be observed in Figure 3.6 and 3.7.

We can see that most variables and the average solution time are correlated. The strongest correlation is with the number of indeterminate states. This does not give us a definitive answer on what variables should be used, but it indicates that they can, to some extent, be used for prediction.

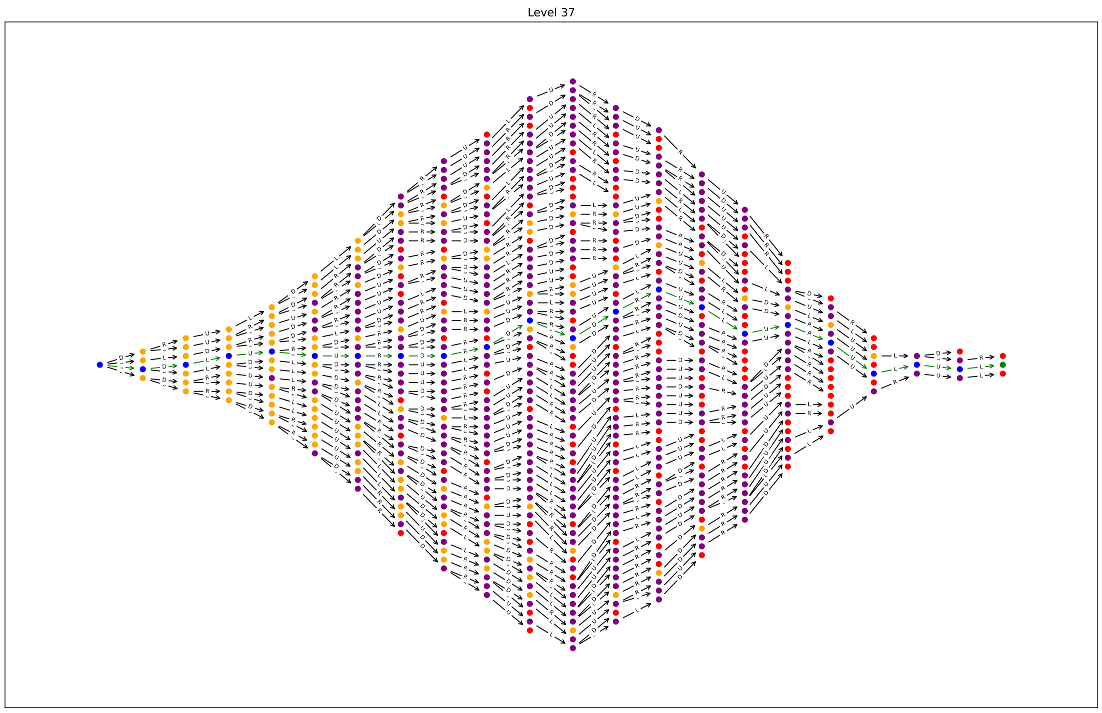

Figure 3.5: Level 37 graph representation.

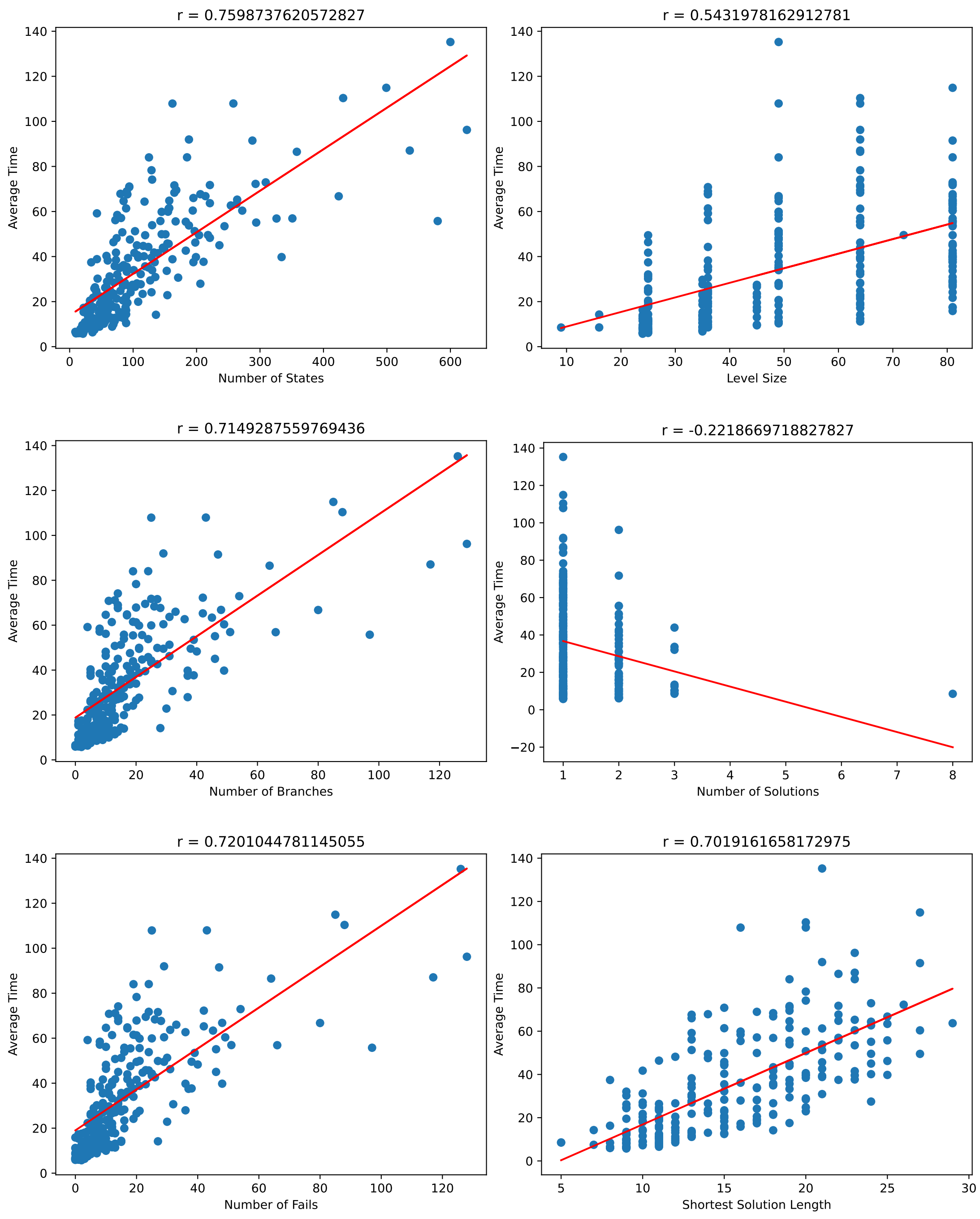

Figure 3.6: Extracted variables plotted to the average solution time for all 250 levels

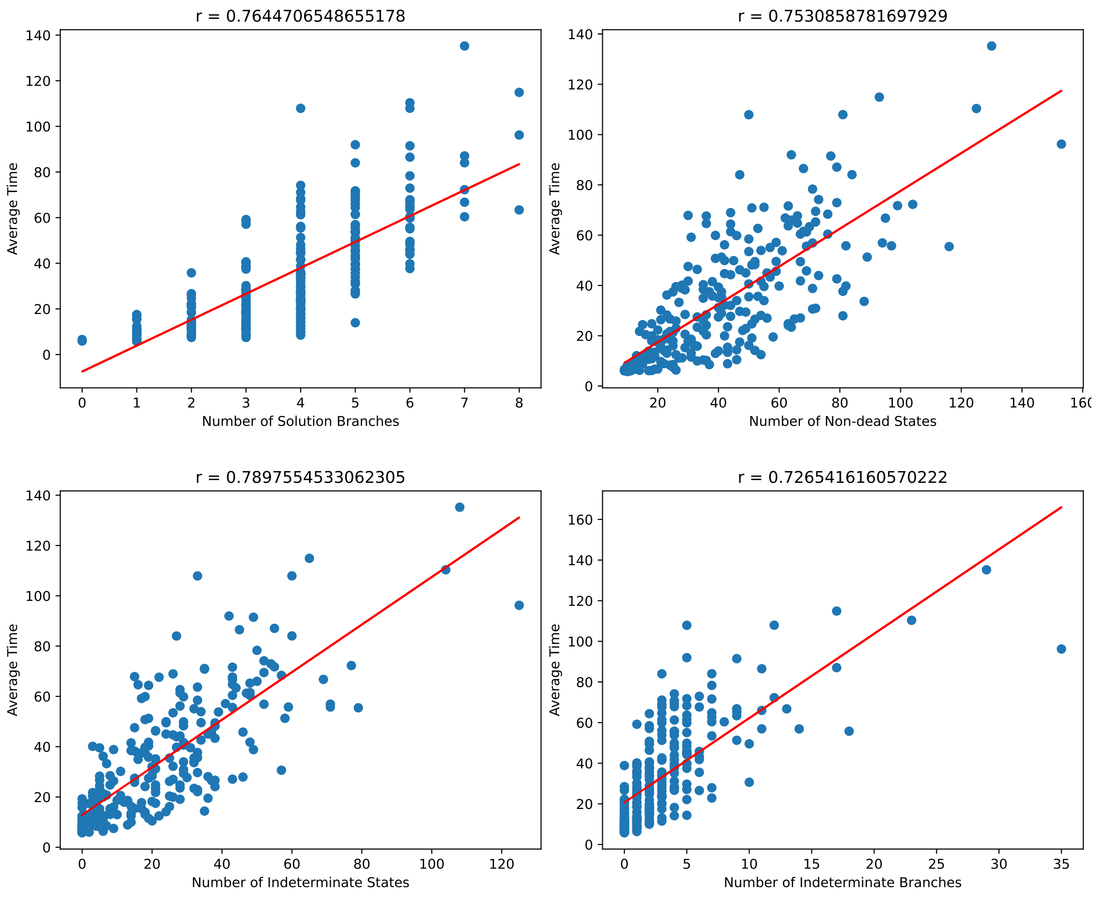

Figure 3.7: Extracted variables plotted to the average solution time for all 250 levels

4 Difficulty Predictor

The method for making the difficulty predictor is based on the work of Kreveld et al. [16] but with some modifications. The goal is to have a predictor that, given a level, can estimate how difficult it is. To implement this, Kreveld et al. used linear regression and focused on selecting a few relevant variables. We built upon this by using other linear regression tools and iteratively selecting what variables to use.

4.1 Tools and method

The script for training was written in Python using the scikit-learn library for the machine learning methods. From this basic linear regression, lasso regression and ridge regression were used. The Pandas library was used for data management. The data consisted of the 250 levels from Longcat with variables derived from their graph representation and the levels' average completion time. There were some deviations between the iterations, but generally, the data was split into 80% training data and 20% test data. Each training was run 50 times with new randomizations of the split, and the Mean Aquared error (MSE) results were averaged over all runs.

4.2 Predictor Development

In this section we describe the development process of the predictor. We go through how each iteration was developed, their problems and how the next iteration was created to solve those problems. A summary of each iteration what method was used, what variables were used, and what MSE it gave can be seen in Figure 4.1.

Iteration 1: Simple Linear Regression

The first iteration was very close to the method presented by Kreveld et al. [16], but without selection. We used linear regression with the first seven variables from the analysis. The resulting model had a MSE of 204.73. Only the first seven variables where used since the others where not discovered yet.

Iteration 2: Simple Linear Regression with variable selection

For the second iteration, we followed the instructions of Kreveld et al. more closely and removed variables that we believed were less relevant. We removed *Size* since it seemed to have a weak correlation. Also, we removed *Number of Fails*. The intuition for *Number of Fails* was that more fails would give a more difficult level, but in practice, it is significantly correlated with *Number of states*, giving very little additional prediction power. The resulting model had an MSE of 199.00. A slight improvement from the previous.

Iteration 3: Lasso Regression

The third iteration used the same variables as iteration 2, but used Lasso regression. To find a good alpha, cross-validation was used. For the cross-validation, the training data was split into 5 random sets, and 1000 was used as the maximum iteration. The resulting model had an MSE of 200.75.

Iteration 4: Ridge Regression

The fourth iteration used the same variables as iterations 2 and 3, but used ridge regression. The same method for finding an alpha value was used as in iteration 3. The resulting model had an MSE of 181.89. This was a substantial improvement, so we continued iterating with ridge regression.

At this stage of the implementation, further analysis was conducted to understand what types of levels the model fails at. During this analysis, we found that some levels had a large number of states but a short average completion time. We played the levels and found that they were easy, as it was clear that certain actions led to a state where a solution was impossible, i.e. dead states. To exploit this understanding, we created the concept of dead states and indeterminate states. Indeterminate states came as a consequence of dead states. Since dead states are states that don't increase difficulty, we really want to measure the number of non-dead states, but this would also include solution states, and a level of only solutions is trivial. Hence, the concept of indeterminate states was created, which is non-dead and non-solution states.

Iteration 5: Indeterminate States

The fifth iteration used the same set-up as iteration 4, but used the variable "Number of indeterminate" states instead of "Number of states." We also found that both the ridge regression and lasso regression minimized "Number of Branches", so it was also removed for this iteration. The resulting model had an MSE of 150.76. This showed a significant improvement, so it could be assumed that the analysis of dead states and indeterminate states is correct.

At this stage, we started work with the generators, using the model from iteration 5. Initially, it was used to categorize levels created by the random generator, and later to train the reinforcement learning methods. However, when playtesting levels, we found unwanted patterns. Some generated levels had many indeterminate states, but were still easy. These levels had "tunnels" that created a long section of indeterminate states where the player could only make one choice. The length of these sections didn't add complexity, but added more indeterminate states, which the predictor evaluated as making the level more difficult. An example of one of these "tunnel" levels can be seen in Figure 4.1. We can see the tunnel with many turns on the left side of the level, and the resulting additional linear indeterminate states in the graph representation. This problem was not present within the original non-generated levels, as the levels were selected not to have tunnels. Most likely, since tunnels just make the level longer but not more fun. To avoid this problem, we chose to use indeterminate branches instead. This solves the problem since branches count states with a out degree larger than one.

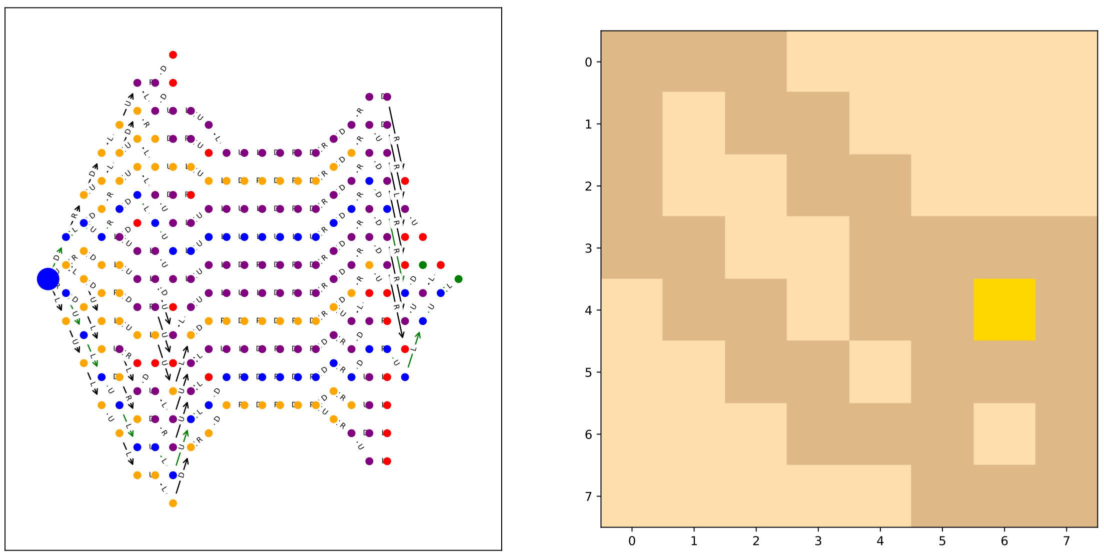

Figure 4.1: Generated level with a long tunnel.

So the tunnels of linear indeterminate states wont add to indeterminate branches while still having the same strengths as indeterminate states.

Iteration 6: Indeterminate Branches

The sixth iteration used the same set-up as iteration 5 but used the variable "Number of indeterminate branches" instead of "Number of indeterminate". The resulting model had an MSE of 166.41. This is worse than the previous model, but the levels generated using this model were perceived to be more difficult when play-tested by us. The play-testing was very unstructured, with us playing levels rated as difficult and seeing if we found them in general more difficult than lower-rated levels. Using this predictor also resulted in fewer tunnels, so it was preferred over the the previous iteration.

Further work with the generators was done, and even though the tunnel problem was reduced, it still persisted. We found that since the variable "Shortest solution length" rewarded the generator for these tunnels, as each state in the tunnel made the solution longer, but it didn't add to the difficulty. We can also see this in the Figure 4.1 as the solutions there have long stretches of solution states, in blue, that would be the same for the player as if there were fewer. The generators could also generate very simple levels that consisted only of a tunnel, but predicted them to be somewhat difficult. To mitigate this, we removed the variable.

Iteration 7: Removing Shortest Solution Length

The seventh iteration used the same set-up as iteration 6 but didn't use the variable "Shortest solution length". The resulting model had an MSE of 187.76. This is also worse than the previous model, but we found that the generated levels were now much more accurately evaluated, as previously over-evaluated tunnel levels were now more accurately categorized as trivial.

Iter.	Method	N States	Size	N Bran.	N sol.	N Fails	Sol. len	N sol. bran.	N ind.	Ind bran.	MSE
1	Linear	✓	✓	✓	✓	✓	✓	✓	✗	✗	204.73
2	Linear	✓	✗	✓	✓	✗	✓	✓	✗	✗	199.00
3	Lasso	✓	✗	✓	✓	✗	✓	✓	✗	✗	200.75
4	Ridge	✓	✗	✓	✓	✗	✓	✓	✗	✗	181.89
5	Ridge	✗	✗	✗	✓	✗	✓	✓	✓	✗	150.76
6	Ridge	✗	✗	✗	✓	✗	✓	✓	✗	✓	166.41
7	Ridge	✗	✗	✗	✓	✗	✗	✓	✗	✓	187.76

Table 4.1: Table showing each iteration of the difficulty predictor, what method was used, what variables were used, and what result it gave.

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.

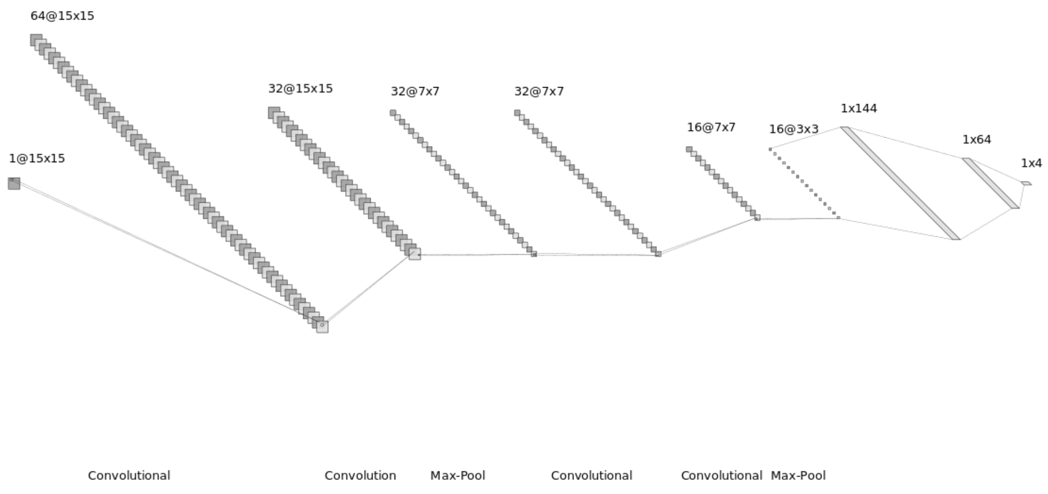

Figure 5.1: Diagram of the convolutional DQN architecture.

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.

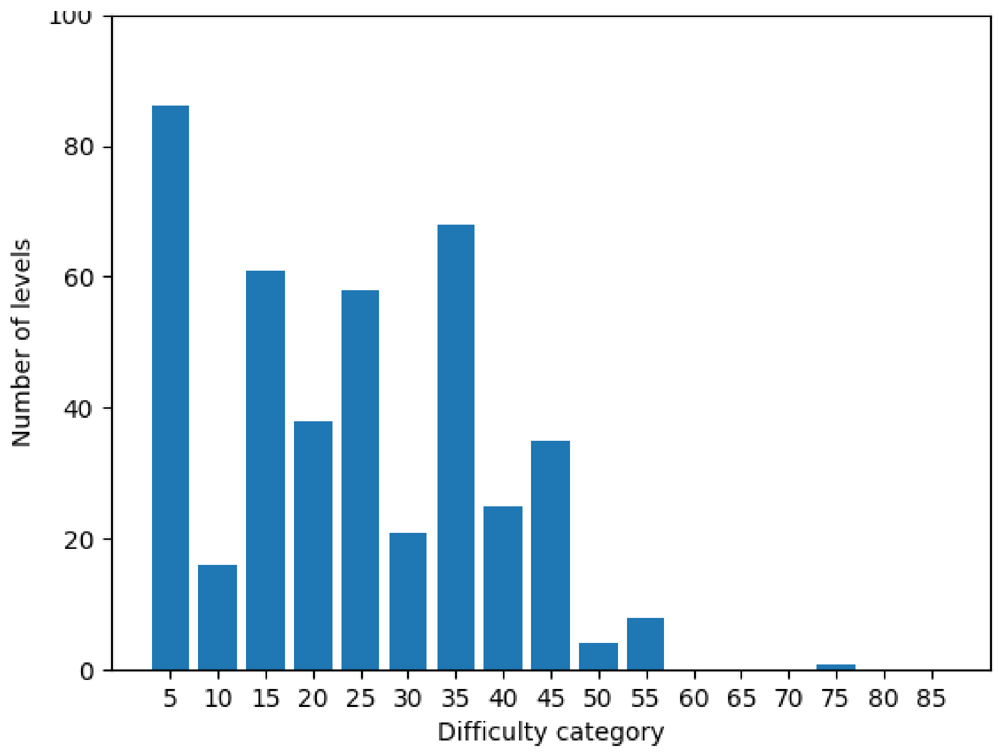

Figure 5.2: Difficulty distribution for unique solvable levels generated by the Puzzle-Up random generator

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.

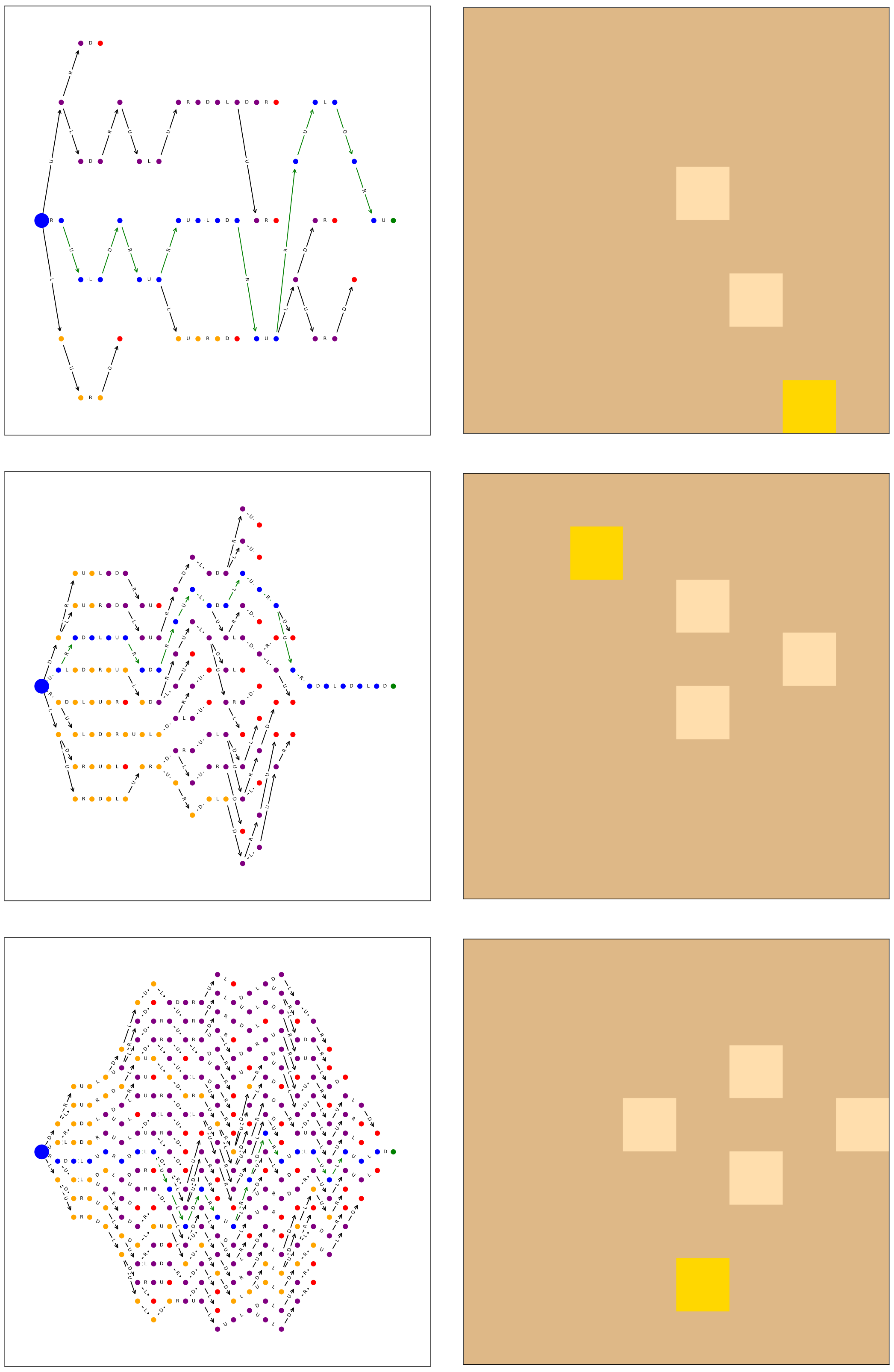

Figure 5.3: Three levels of varying difficulty sampled from the ones generated by the Puzzle-Up random generator.

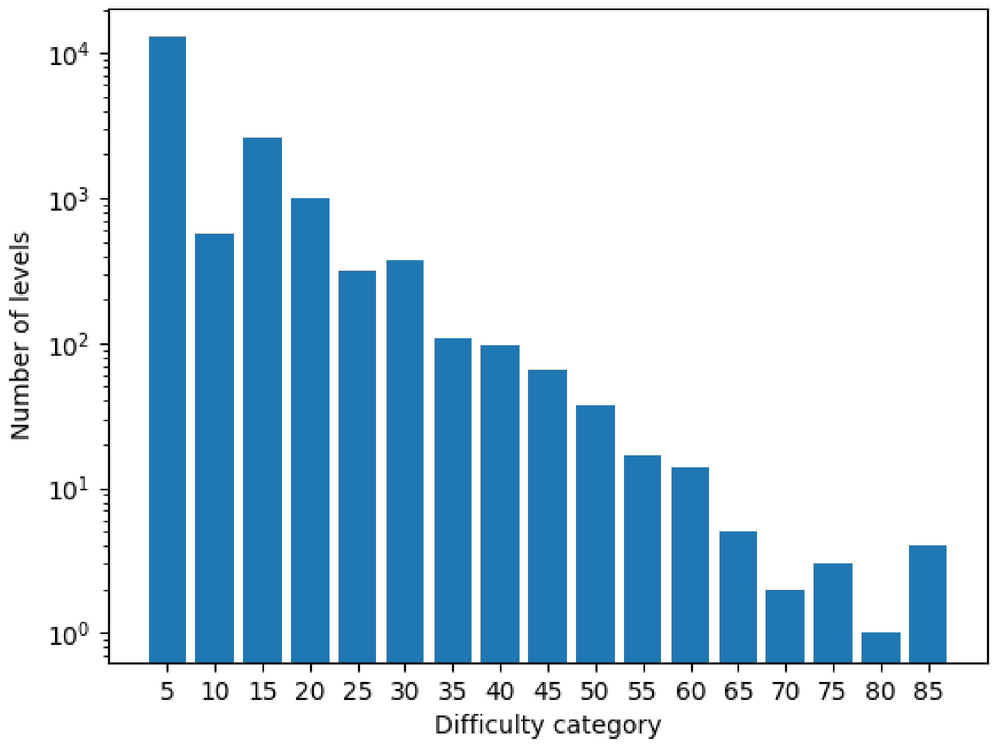

Figure 5.4: Difficulty distribution for unique solvable levels generated by the Solution-Down random generator. The y-axis is logarithmic.

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

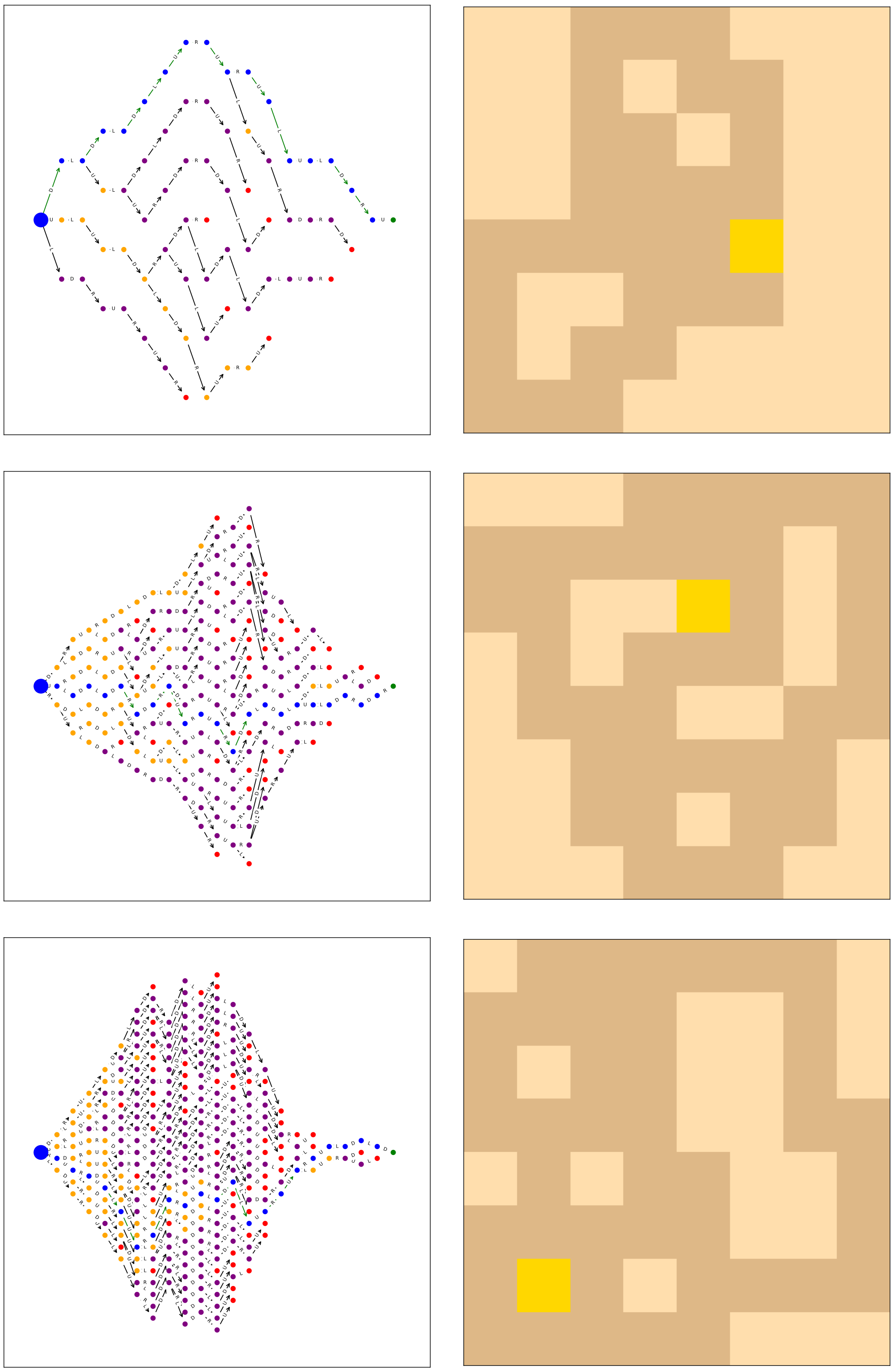

Figure 5.5: Three levels of varying difficulty sampled from the ones generated by the Solution-Down random generator.

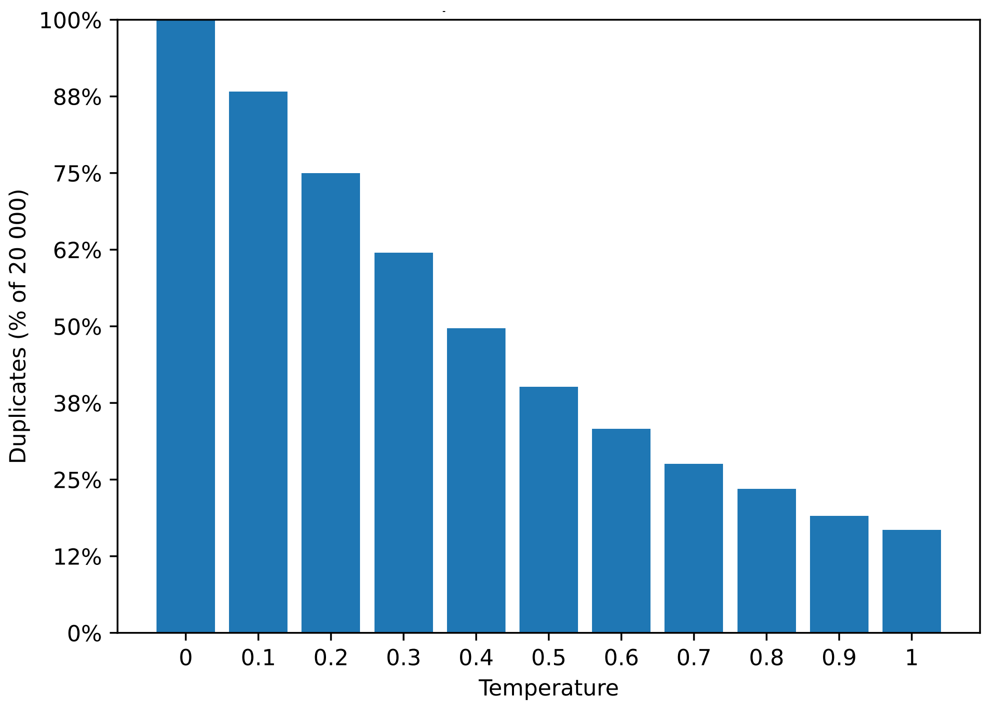

Figure 5.6: Number of duplicates for each temperature of generation by the SD-PCGRL generator

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.

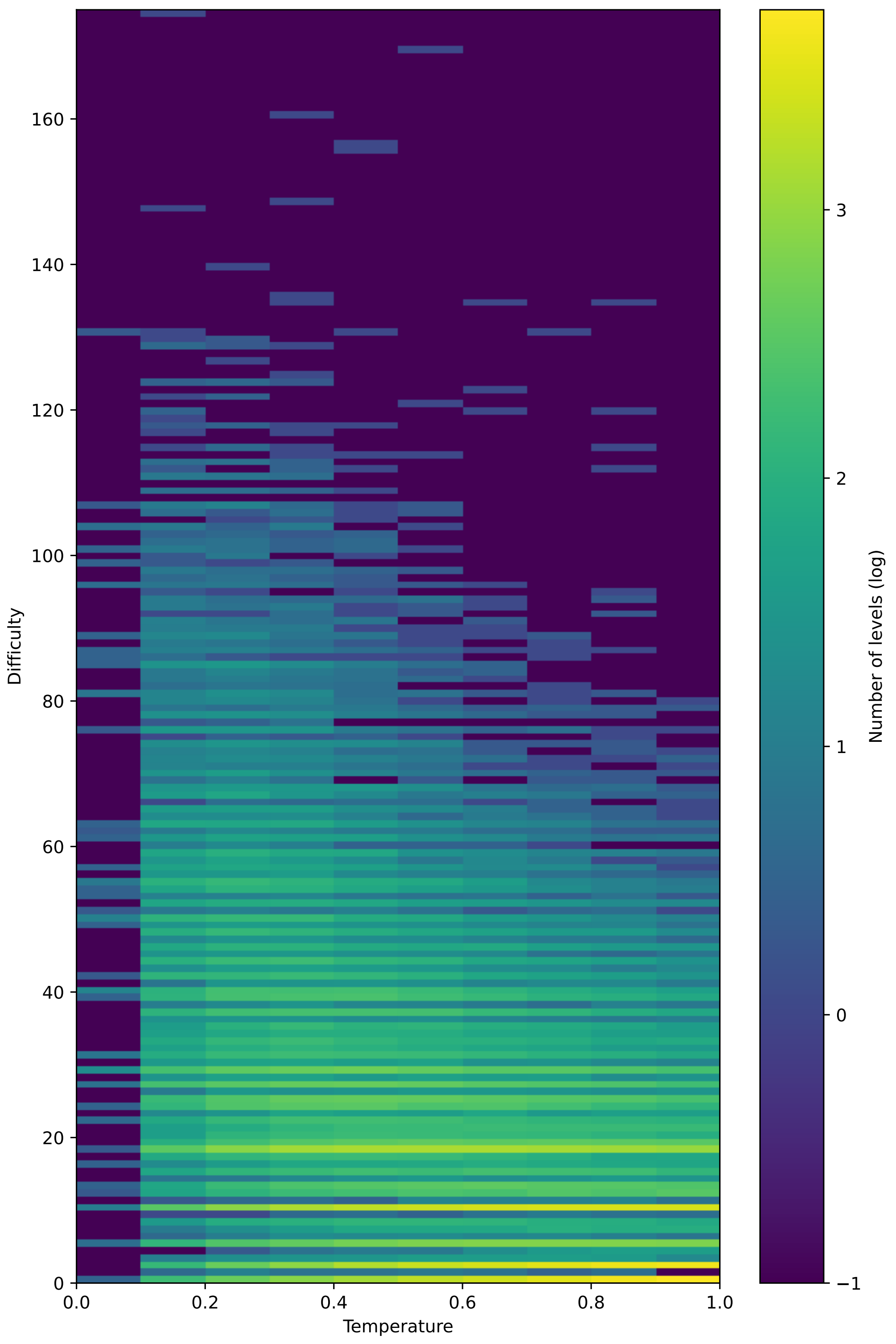

Figure 5.7: Number of duplicates for each temperature of generation by the SD-PCGRL generator

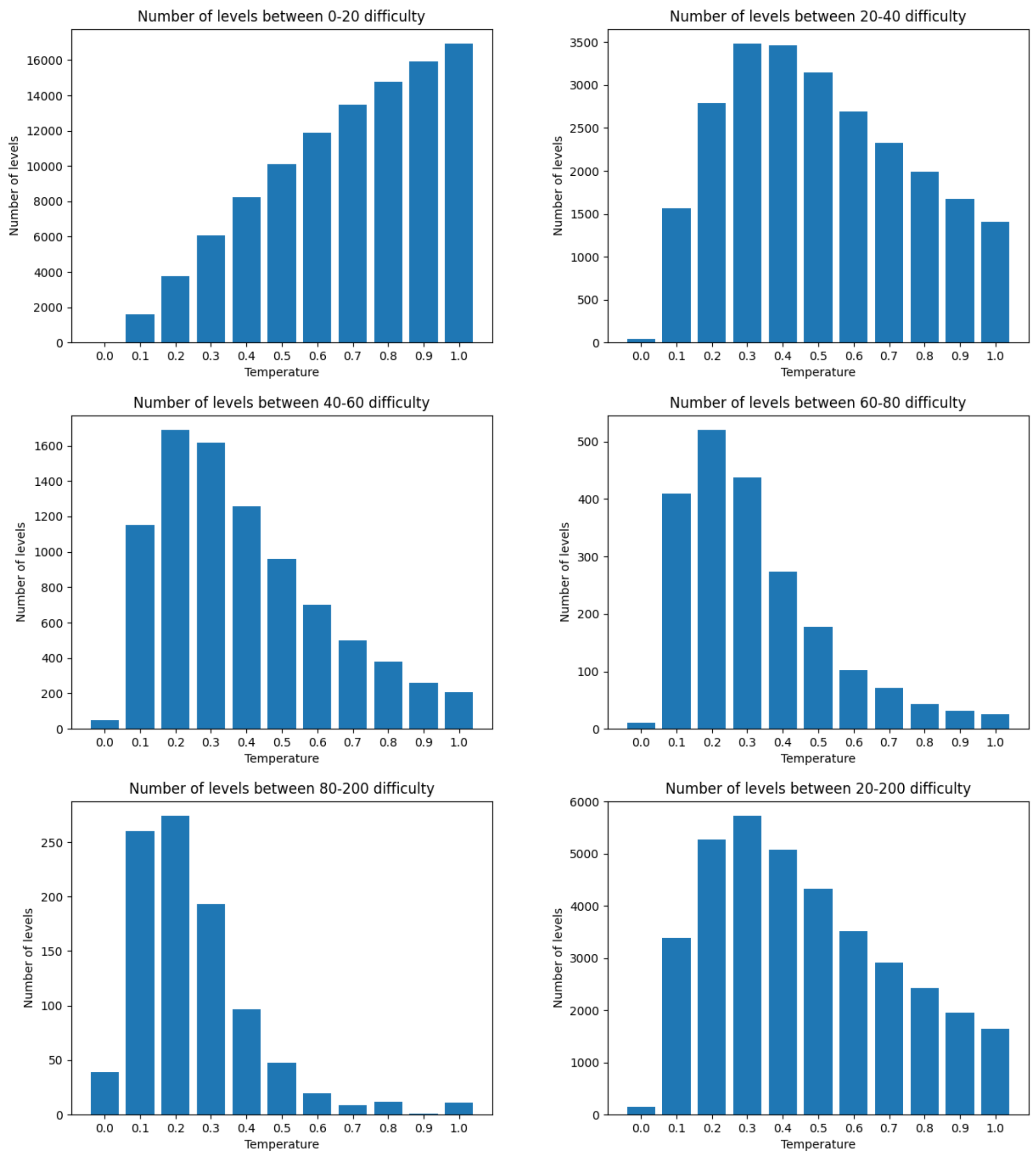

Figure 5.8: Number of levels generated by the SD-PCGRL using all temperatures, for various difficulty spans.

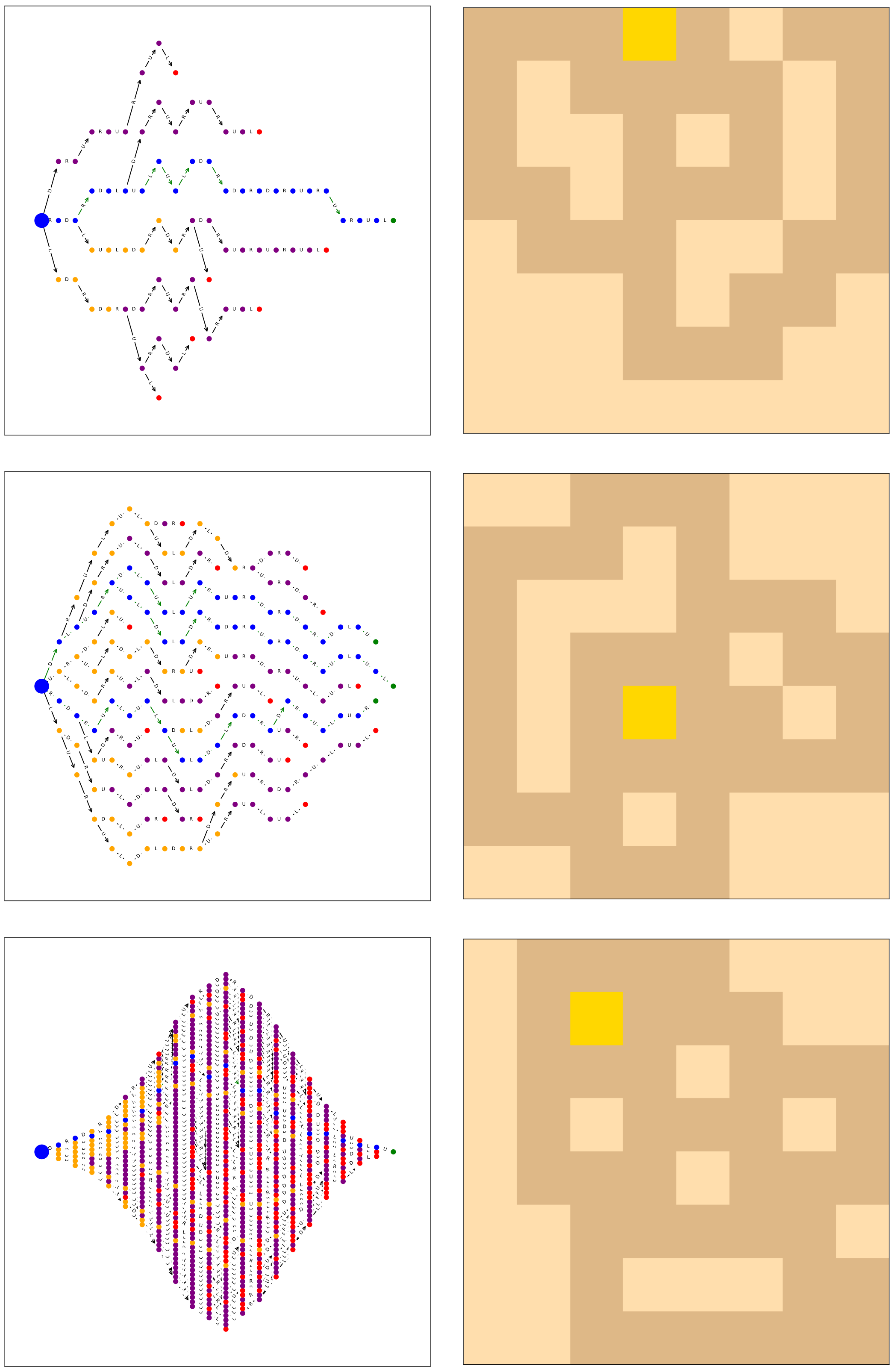

Figure 5.9: Three levels of varying difficulty sampled from the ones generated by the SD-PCGRL generator with temperature 0.3.

6 Discussion

This chapter discusses the methods and results used in the report. The chapter is divided into analysis, predictor, and generator sections, which examine their corresponding chapters in the report. We also have a suggested approach section where the entire development process as a whole is examined. After this, we also discuss potential future work that could be done and how this work fits into a broader socio-cultural context.

6.1 Analysis

The analysis mainly consisted of an explanation and use of the graph representation, and a presentation of the data. This section will discuss both. Going into the strengths and weaknesses of the graph representation, as well as future uses of it, and discussing the data compared to previous work, as well as its possible faults.

Graph Representation

The graph representation was an important part of the analysis and essential for developing the predictor, but it is not directly relevant to the research question in this work. It is still an exciting part of the work conducted for this report. Therefore, this section will discuss what we found useful with the graph representation, how it could be used to model player behavior, its weaknesses, and how it could be used in other games, even though these subjects are irrelevant to the research questions.

As a design tool

The graph representation was developed to gain insight into the structure of Longcat levels, since, unlike other casual puzzle games such as the one analyzed by Kreveld et al. [16], Longcat has almost no puzzle elements to observe. This led us to look at actions and states instead and playing the game as a way of searching through the graph to find a solution. For us, this turned out to be a strong design tool. At first, simply looking at the graph size was a decent indicator of difficulty, an intuition supported by the data as shown in Figure 3.6 and 3.7, but as we found exceptions to the rule, we could analyze the graphs more to gain further understanding. This lead to concepts such as dead-states and branching. So aside from being a good way of extracting information from the levels, the solution graph also helped us improve our understanding of the game.

These insights could most likely be found through a more traditional approach of having a designer design and test levels, but what we gain beyond that is also a way of quantifying these otherwise loose aspects of level design.

For modeling player behavior

The concept of dead states and indeterminate states, especially, is fascinating. By adding them to the graph representation, we are modeling not only what actions and states exist in gameplay but also how we believe the player will analyze each state. The idea being that players will immediately see that a dead state is unsolvable and will not explore states beyond that point, but restart instead. Indeterminate states are states where the player can get lost since it isn't evident that they will not lead to a solution. This fits quite well with how we have observed players to solve Longcat puzzles during unofficial testing, and might also apply to puzzles in general. The player explores some actions until they find that the puzzle is unsolvable, in which case they start over. This repeats until they find a series of actions that lead to a solution. It is clear, then, that if there are more states and branches in which the player can explore without reaching a solution, then the level will be more difficult as the probability of selecting a solution branch decreases. However, this somewhat assumes that the player selects actions randomly, but what sets more experienced players apart seems to be how early they can recognize a state as unsolvable, and what actions they choose to select first. This action selection is probably based on a combination of player knowledge and an intuition built from playing the game. Aside from creating more states and branches for the player to search through, we could also challenge the player's knowledge and perhaps subvert the player's intuition.

Weaknesses

Representing the player as an agent searching through the solution graph is useful for predicting difficulty, as we can see from its correlation with the data in Figure 3.6 and 3.7 and the resulting predictor in chapter 4. Some data supports it, but it still needs more work to be used as a reliable way of modeling player behavior in the way we discussed above. It also has some clear problems.

First, the process is heavily based on a designer analyzing how the player interprets the game. This means it needs a lot of legwork, but also that the designer might be incorrect in their analysis, miss some crucial variables, or be otherwise biased in their observations. Second, we lose information on how the level is represented visually to the player. How information is conveyed plays a significant role in a player's actions.

Use in other puzzle games

As we discussed, graph representation might be useful in several aspects of game design, but how applicable is it to other puzzle games? And what type of puzzle games is it useful for?

The obvious puzzle game categories that becomes difficult to model with the graph representation is non-discrete and non-deterministic games. The graph representation is based on discrete states and actions, so if the game has non-discrete states, we must somehow discretize it. This is possible, but not trivial, and we may lose some information in the process. Similarly, we can model nondeterminism in the graph, but it is not trivial and may complicate the analysis of the graph.

Unsurprisingly, games similar to Longcat seem to be the best candidates. One game that is relevant for its similarity but also presence in research within the area of puzzle games is *Sokoban*. This game has been discussed in related works and revolves around pushing boxes into specified positions. The state in *Sokoban* is decided by the position of the player character and the boxes. The actions in *Sokoban* are the four directions the player can move in. This means that a relatively simple level, such as one with size 8x8 and two boxes, could potentially have $62 * 62 * 62 = 23,8328$ different states. This is much larger than 1151, the largest number of states we have observed for any generasetd level in Longcat, and too many to effortlessly process in the way we have analyzed Longcat levels.

This difference comes from the fact that Longcat constrains the player more in terms of what actions they can take and how actions affect the game. It also has fewer puzzle elements that can expand the number of states. In *Sokoban*, the player can most of the time move as they please, and there are several boxes, but in Longcat, the player usually can only move in one of two directions, there is only the player in the level, and their actions give less liberated movement. This results in Longcat having fewer total states, which makes it well-suited for the graph representation. *Sokoban* might be too complex a game for its application.

A possible solution to this could be to look at *meta-actions* to reduce the number of states in the *Sokoban* representation. *Meta-actions* could here be that instead of looking at the player's movement as an action, we look at what boxes they could push as their possible actions. This would substantially reduce the number of states, but it still might not be enough.

From this *Sokoban* example, we can see that applying the graph representation to other games may not be trivial. It might be easier, more useful for other games, or possible with some simplification, such as *meta-actions*. The large number of states might also not be as large an issue as we believe. Either way, more work would have to be done to fully understand its potential usefulness.

Data

In this work, we used real player data from Longcat. This data was sampled from 12,000 players, giving an average solution time for 250 levels. We believe this large player count provides a reliable average for each level. We also believe that using player time gives a more empirical result that isn't influenced by the player's perception. This is in contrast to Kreveld et al. [16], which used a small number of players and player-reported difficulty.

However, this did result in us having to make the assumption that average solution time is correlated with difficulty. We find this to be a valid assumption. If a level is more difficult, it should take longer to complete, but other factors could increase the solution time. Some levels may not be difficult, but might need a long series of actions to complete, which increases the solution time. Since the data also comes from real players, there may be some noise in the data from players being distracted while playing, leaving the game on while they do something else, or some other act that could add to the solution time.

Despite this, we still believe this to be a reliable measure of difficulty. The number of actions required to complete a level is an issue we observed later in the work and accounted for when making the predictor, as we will discuss in the next section. As for the possible noise in the data, we believe that distracted players and other outliers are a small minority, and since we have such a large number of players, they shouldn't greatly affect the average.

A larger problem with the data is that the average calculated for each level consists of different numbers of players and various types of players. This is caused by Longcat's level progression. A player must play the levels in order and complete a level to reach the next one. So, later levels will have fewer players and thus less data. This can be seen in Figure 3.4. This also means that later levels will have players who are more experienced with the game and more likely to have a higher interest in puzzle games such as Longcat. This could result in earlier levels' difficulty being overestimated and later, once being underestimated.

This problem could have been solved by removing large outliers and averaging only from players who have completed all levels or for fewer levels. The second solution, however, was not possible since the player IDs in the database only tracked play sessions, so one player could not be identified over all levels.

This touches on another problem with the data. The database was set up in a questionable way, making it difficult to extract important data and impossible to track certain player statistics. The database also spans multiple versions of the game throughout its lifetime. It is possible that this has caused some unknown anomalies or noise within the data.

6.2 Difficulty Predictor

The chapter on the difficulty predictor covered the iterative development process of the predictor. In this section, we will discuss this methodology and the result from the predictors. We will also discuss the variable selection process and why we believe some variables proved more useful than others, which will answer research question 1 as well as we can.

Method

The difficulty predictor itself is not what we wanted to explore in this work, but it was an important component to explore the generation methods. Therefore, we didn't focus on a replicable method but instead on one that would give an acceptable result that could be used for the generators. So we used an iterative approach where we could change models and parameters for what we believed would give a better result, and pause the development with further analysis. This also allowed us to change the difficulty predictor later when we found that the generator could abuse it to gain undeserved rewards.

It did, however, lead to a less reliable result. We can not say for sure that this is the best predictor possible with the variables we found, and it could be difficult to compare in future work. It is also very likely that we could have achieved a better result using a method different from linear regression. Some variables might not have a linear correlation with difficulty, and so a method that could model this relation could be better.

Results

The only clear result is the MSE of the models. This is a good comparative result, which was useful for comparing models during development, but it doesn't give a clear answer on how good the predictor is. We also made some debatable choices during development that increased the MSE, such as replacing indeterminate states with indeterminate branches and removing the shortest solution variable. The only motivation for this is that the generator could abuse these variables to gain undeserved reward, but if these variables could be increased without increasing difficulty then why did they give good results?

We have two theories. First, we are not tracking difficulty directly, but the average solution time for a level. These tunnels that the generator creates may not increase difficulty, but they do increase time to solve it since more actions are needed to solve the level, and each action has a short animation that adds time. Second, the levels in the data are created or selected by a designer, so they do not contain any long corridors since they aren't fun. If the data had included many linear corridor levels, where the player can make few choices but needs to do many actions, then the completion time for these levels would be lower then an actually difficult levels, and solution length and number of states would have been worse predictors.

It is also possible that these average solution time and indeterminate states do affect difficulty, but the data or some aspect of the model allows the generator to exploit it. A likely cause is that we are using a simple linear model. A level with a long solution but no branches is trivial. However, it might be the case that a level with a lot of branching is more difficult with a longer solution than a shorter one. This relation is impossible for a linear model to map. We may have gotten a better result and a less exploitable predictor if a more sophisticated model had been used.

We still believe that the predictor we used was good enough. During the generator's training, we found out that we didn't need a perfect difficulty predictor but a predictor that worked fast, rewarded the generator for changes that made the level more difficult, and didn't reward the generator for unwanted level aspects, such as corridors. The final predictor does this by looking at some variables that we are relatively confident the generator can't exploit and that affect difficulty; this is good enough. Almost more crucially, it does this very quickly, which allows for faster training.

Variables

After several development and analysis iterations, we analyzed eleven different variables in total. Most of them were not used. The ones we found most relevant for predicting difficulty were number of indeterminate branches, number of solutions, and number of solution branches. We also found number of indeterminate states and shortest solution length to be relevant but were not used for reasons described in the previous sub-sections.

We believe number of indeterminate branches increases difficulty since it counts the number of states where the player needs to choose without clearly knowing if the level is solvable from the current state. It means more possible paths to search through until the solution is found. It is important here that we don't count dead states, where it is clear that the level is unsolvable, since the player will simply restart at such a state.

We believe number of solutions decreases difficulty since more solutions make it likelier for the player to find a solution when searching different paths. This is more so true for smaller graphs, but as the graphs grow larger, there are rarely enough solutions to outweigh the large number of indeterminate branches.

We believe number of solution branches increases difficulty since it counts the number of possible incorrect actions the player could make while on a solution path. If, for example, we had a level that started with one branch, the first leading to the solution after some actions and the other leading to a very complex graph, then the level would still be simple since the player just has to guess correctly once at the beginning of the level to find the solution.

Why were the other variables less relevant for prediction? As for size, we believe it wasn't relevant enough for predicting difficulty; the rest we believe were relevant, but only because they were correlated with number of indeterminate branches. We can somewhat see this during the variable selection process. First, we removed number of branches and number of fails since they were so closely correlated with number of states, which was our strongest predictor at the time. We then further refined number of states by defining dead states and subsequently number of non-dead, which eventually led to number of indeterminate states. We then choose to use number of indeterminate branches instead of number of indeterminate states for reasons we have discussed. The variable selection process has then been a refinement process to find the underlying variable that actually increases difficulty, which many variables are correlated with.

Our selected variables may also be correlated with a more appropriate underlying variable that we have not found, and we may have lost some predictive power during the variable selection process. Most likely in that case, in the step from number of indeterminate states to number of indeterminate branches. This is, however, as we discussed in the previous subsection, good enough for the purposes of this work.

6.3 Generator

In this section, we will discuss the generator chapter.

This section is divided into four subsections, each discussing one result metric from the generator tests and attempting to answer one research question: a solvability section, which will discuss research question 2; a difficult distribution section, which will discuss research question 3; a duplicates section, which will discuss research question 4; and a difficulty span section, which will discuss research question 5.

Solvability

As we have described, the solution-down generators, SD-PCGRL and the solution-down random generator, guarantee solvability from their generation process, so they have 100% Solvability. So in this section we will focus on the solvability of the levels generated with the generator based on the original PCGRL turtle method and the puzzle-up random generator. Both these generators can be seen as types of puzzle-up generators. The puzzle-up random generator had 1107 of 20,000 iterations that resulted in at least one solvable level. That is 5.54% of iterations resulting in solvable levels. This is also after the generator was improved from the original used in the application. The generator based on the Turtle PCGRL was unable to create a single solvable level, resulting in 0% solvability. These scores are very poor, which means both generators are having problems with generating solvable Longcat levels. This is even despite the random generator being improved from the one in the app, as described in its implementation section, and the PCGRL generator being trained with some assistance to find solvable levels in its reward function.

The Turtle PCGRL method has worked well compared to other methods for generating Sokoban levels [18]. The random generator is also used in Longcat with acceptable success. Why do they have such poor performance? As for the Puzzle-Up random generator, it is implemented and limited in such a way that it will work for the application, so this result is to be expected. However, the turtle PCGRL generator failed spectacularly when it succeeded for other games. We believe it has to do with how harsh the limitations for solvability are in Longcat. This solvability limitation means a tiny subset of all possible level states count as solvable levels. This generally makes it difficult to generate levels with a puzzle-up approach. The turtle method here can be seen as an MDP with movement actions and changing the current square actions. In the turtle MDP, only a subset of states result in a solvable level for any game it tries to generate levels for, and how small this subset is will be different from game to game. Furthermore, the number of states and actions will depend on the number of square types in the game. We believe that the MDP, when using the turtle method for Longcat, is too complex and has too small a subset of solvable levels for the agent to learn how to create a solvable level.

How is the solution-down environment different? The solution-down environment also defines an MDP, but one with fewer actions and where every state is a solvable level. This helps the agent substantially. It no longer has to learn what makes a level solvable, and each state has a smaller degree, making it simpler to search through and learn from. The advantages of solution-down are extremely apparent for Longcat, thanks to its harsh solvability limitations, because of this, it is unclear how useful it would be for other games. Still, we also believe it can have advantages there as well. As long as there are some unsolvable levels in the MDP, the process can be improved by removing them and having the agent instead focus on learning qualitative aspects of the game. However, this essentially involves designing a unique MDP for each game, and we do not know if it is possible for all games, nor do we have a suggested workflow for developing such specialized MDPs.

Difficulty Distribution

What we are looking for is a well-spread-out difficulty distribution, so we can generate levels of varying difficulty. Turtle PCGRL did not generate any solvable levels, so it did not give any difficulty distribution. Figure 5.2 shows the puzzle-up random generator difficulty distribution. It didn't generate a large number of solvable levels, but the distribution itself is good with a consistent number of levels from difficulty 0 to 50, which are easy to medium difficulty levels. There are very few challenging levels, with only one at difficulty 75. This distribution is a somewhat unreliable since it only consists of 421 unique levels.

Figure 5.4 shows the solution-down random generator's difficulty distribution. This distribution is more lopsided. Almost all levels generated are of a trivial or easy difficulty, but there are some of higher difficulty. The generator also reaches a higher difficulty than the puzzle-up random generator, going as high as 85 difficulty, and generating eight levels of difficulty 75 or higher. Despite the distribution not being as even, the generator still gives more levels of varying difficulty from the sheer number of levels generated.

Figure 5.7 shows the SD-PCGRL generator's difficulty distributions at different temperatures. A higher temperature means more randomness, so a temperature of 1.0 is equivalent to the solution-down random generator. We can see in the graph that as the temperature decreases, the distribution becomes more spread out, until 0.0 is reached, where the distribution is worse again. This answers research question 3. SD-PCGRL has a more spread-out distribution at lower temperatures than the ones generated with random generators.

Duplicates

The puzzle-up random generator had 759 duplicates among 1107 generated solvable levels, giving us 68.56% duplicates. On top of the low solvability, this means the generator gives us as few as 348 unique solvable levels overall, or 1.75% of generated levels.

The solution-down random generator generated 1936 duplicates among 20,000 levels, giving us 9.68% duplicates. This is a small percentage compared to the puzzle-up generator, but also generators from previous work. Why this number is so low could probably be attributed to the fact that the environment ensures solvability, so the generator can search through more solvable states, allowing it to find more unique levels. The puzzle-up method is limited in how deep it can go, since it seldom finds solvable levels when too many walls are placed. So most levels it finds have only a small number of walls and so there will be fewer levels for it to find.

The number of duplicates for the SD-PCGRL generator at different temperatures can be seen in Figure 5.6. As can be seen, the number of duplicates decreases with higher temperatures, the highest having the same number of duplicates as the solution-down random generator. This is to be expected, since a lower temperature will result in the generator being more deterministic, so it will be likelier to recreate the same levels.

This allows us to answer research question 4. SD-PCGRL has a similar number of duplicates as the solution-down random generator at higher temperatures and substantially more duplicates at lower temperatures. SD-PCGRL has more duplicates than the puzzle-up random generator at temperature 0.3 and lower.

Difficulty spans

We have examined solvability and duplicates to follow the methodology of previous work and gain a good understanding of the number of usable levels generated. We have also examined difficulty distribution to determine the variety of difficulties generated and find what generator is most suitable for generating an engaging difficulty curve. However, we have found that the SD-PCGRL at temperatures 0.1-0.3 gives a better distribution than the solution-down random generator, but has substantially more duplicates. So is the random generator better since we get more levels, even if it has a worse distribution? We need to look at the absolute number of generated levels at different difficulties to understand which gives a larger variety of level difficulties. This is why we decided to observe difficulty spans for the SD-PCGRL.

We can see the results of the difficulty span analysis in Figure 5.8. For the first difficulty span from 0 to 20, we can see that the temperature 1.0, which is equivalent to the solution-down random generator, has the most significant number of generated levels. Important to note is that the random generator generated 16924 levels of this difficulty. With 18570 unique generated levels, this means 91.1% of all the unique levels generated with the solution-down random generator fall within this difficulty span. We believe levels in this difficulty span are essentially unusable. To illustrate this point, we sampled three levels of varying difficulties within the span 0 to 20 generated by the solution-down random generator. These levels can be seen in Figure 6.1. Looking at them in order from top to bottom, we can see that the first is unfailable. The rest have some branches from the starting state, one or two leading directly to a solution and the other to paths of dead states. It is trivial for most players to understand that these paths do not lead to a solution. These levels could prove interesting for a novice player, but we believe they are trivial and uninteresting for even moderately experienced players.

If we observe the other difficult spans in Figure 5.8, we can see that the temperatures 0.1 to 0.3 have the most generated levels for each span. The comparative difference also becomes larger as we look at spans of higher difficulty. As stated in the result, for the difficulty span 80 to 200, the temperature 0.2 has generated 24 times more levels than the solution-down random generator. This makes it clear that SD-PCGRL at a temperature around 0.2 is better at consistently creating more difficult levels.

This data and analysis also allow us to answer research question 5. Levels of difficulty less than 20 as trivial are categorized as trivial. In that case, the total number of non-trivial levels generated by the SD-PCGRL at each temperature can be seen in Figure 6.2. From this, we can see that the SD-PCGRL at its best generated 5727 non-trivial levels at temperature 0.3. This is 28.64% of all its generated levels. In comparison, the solution-down random generator generated 1648 non-trivial levels, which is 8.24% of all its generated levels.

From this, we can see that among the generators tested SD-PCGRL has been shown to be best for generating unique, solvable, and non-trivial levels of varying difficulty.

6.4 Future work

This section discusses what future work could be done to build upon or work here. We will first explore how the SD-PCGRL could be used as an approach for developing level generators for other games, and what games this could apply to. Second, we will discuss how the generator could be improved.

Suggested approach and other games

In this work, we have combined a novel approach to content generation for games, SD-PCGRL, with a difficulty predictor to create a generator that could generate levels that engage and challenge players. How could this approach be used for other games? Two components are needed: a solution-down generation method and a difficulty estimator.

We have no generalized way of designing a solution-down generation process. Intuitively, it should be possible for most games with a goal and some actions, but the effectiveness of the process will likely vary depending on the design and game. It is certainly possible that Longcat is specifically well suited for using a solution-down generation process and that there are clearer downsides in other games. More work would be needed to verify how useful solution-down generation processes are for other games. An interesting area of study would be their application to non-puzzle games. We believe solution-down processes can be applied to action games such as platformers. This could work by seeing a series of inputs at different time intervals as the solution to a platforming level. If the game's physics are deterministic, then obstacles could be generated from a series of inputs that would solve/complete the level. This would be very interesting since, as we understand it, the procedural generation of platform levels is a complex problem since it is difficult to verify their solvability. Solution-down generation could then be a solution to this.

The difficulty predictor has a similar problem, as the complexity of automatically predicting difficulty for a game is dependent on the game itself. These tools are, however, more well-researched and already used in game development, so depending on the developer, it might not be a significant challenge. As discussed before, the graph representation could assist with this for some games, but probably mostly for puzzle games.

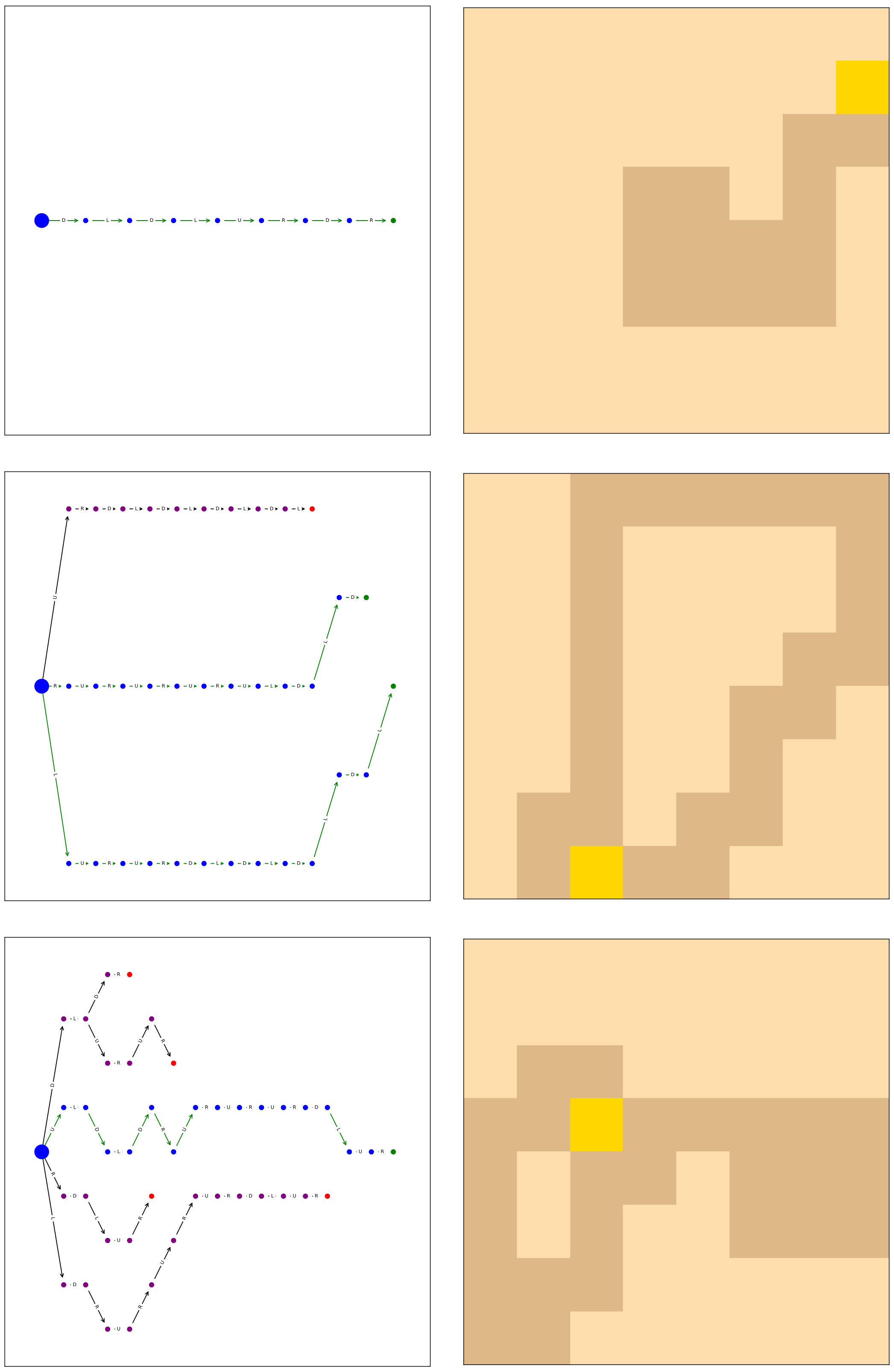

Figure 6.1: Three levels of varying difficulty in the span 0 to 20 sampled from the ones generated by the solution-down random generator.

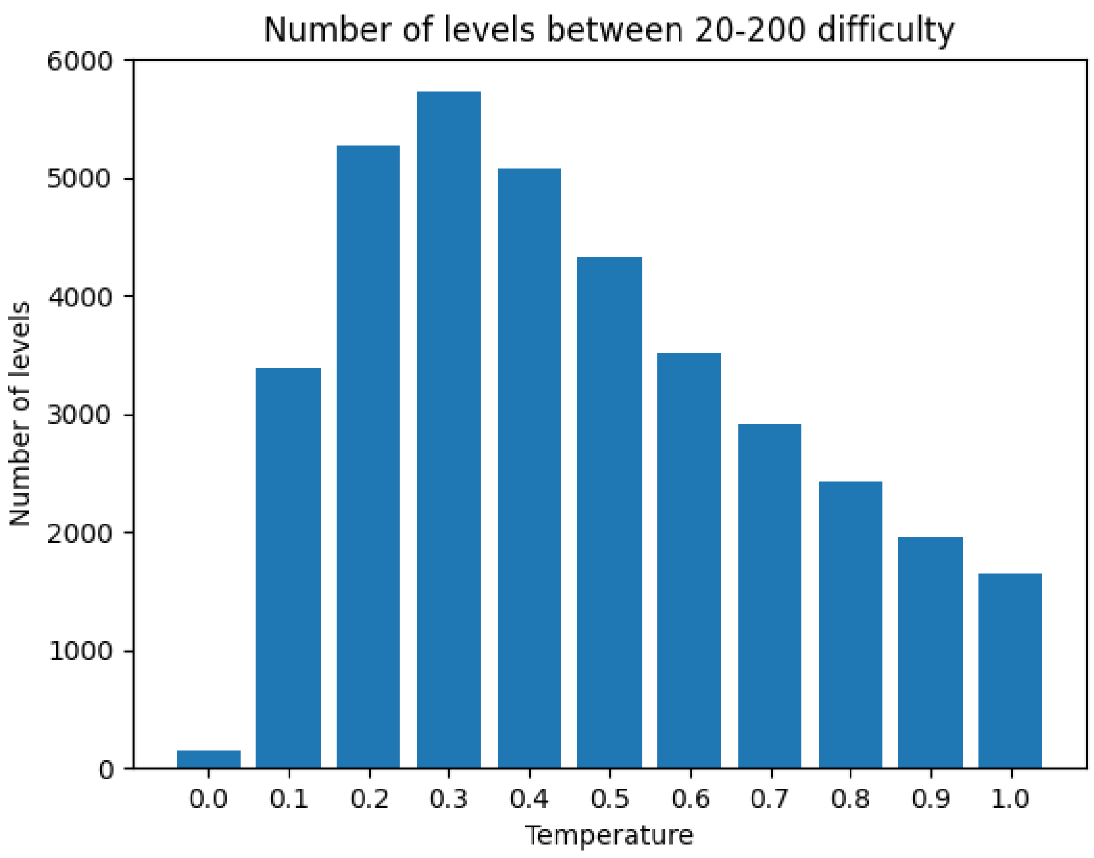

Figure 6.2: Number of non-trivial unique levels generated by the SD-PCGRL generator at each temperature.

So the suggested approach requires a lot of specialized work for each game. In this way, the original PCGRL is superior as it proposes a generalized approach, but as we have seen, it can have problems for games like Longcat. Some parts of the approach could be generalized, but that needs further work. However, in an actual development environment, it is not unreasonable for a designer to design both the solution-down generation process and difficulty estimator if it can give a more reliable level generator. A good designer would also have enough domain knowledge about the game to design these components well.

Improving generator

There are plenty of possible improvements for the SD-PCGRL generator. First, it could be improved by training it for longer. In our research, we limited the training episodes because of time constraints, but it seems from the training tests that it could be improved. Improvements could also be made to the DQN architecture and training parameters. We selected these iteratively during development, and a more thorough work could probably find a better setup.

Another improvement would be having the generator take a target difficulty. For now, we have focused on having it try to reach as high a difficulty as possible and used randomness to get easy levels, but if a model could directly find a level of given difficulty, it could speed up the generation process significantly.

In the same vein, it would be interesting to see what other aspects of models besides difficulty the generator could learn to produce. Perhaps the number of walls, turns, or some other aspect can create variation for gameplay or visually.

Last, it would be interesting to "close the loop" by serving players generated levels and using the eventual player data to improve the generation process. This could be done to generate a target difficulty more accurately, or it could be used to track engagement more directly by attempting to create levels that will keep the player playing, assuming that playtime is more directly connected with engagement.

6.5 Wider context

In this work, we have looked at using machine learning for procedural generation in games, which is ultimately a form of entertainment. Games are enjoyed by millions each day, and entertainment could arguably be described as something essential for many people's lives.

However, there is a darker side to this. Many modern applications implement machine learning for recommendation algorithms designed to keep the user on the application for as long as possible. It is still unclear what consequences the frequent use of these applications has, but they have been shown to have negative effects on users' mental health in some situations [17]. We are looking at engaging the user, but how is this different from attempting to keep the user on the application for as long as possible? Are we in the process of making an equivalent hyper addictive game? There seems to be some boundary where engagement goes from being something positive that makes the user like the application, to something detrimental that promotes addiction and negatively affects the user. We do not believe our work will have that effect on users, but using machine learning to keep the players' attention in this way may eventually lead to it.

With that said, games are not purely negative for users. Puzzle games specifically have been associated with reduced risk of dementia, [5], and making them more engaging by further challenging the player could help people keep the habit.

7 Conclusion

In this work, we introduced and evaluated a novel approach for content generation for games, which we call solution-down procedural content generation via reinforcement learning (SD-PCGRL), by using it to generate levels for the hyper-casual puzzle game Longcat. We also generated not only solvable levels, but also made them engaging and unique. We chose to define engagement around challenging the player, meaning that we aimed to find a generator that can generate levels of a variety of difficulties. We also compared SD-PCGRL to three other generation methods: an adapted version of the original PCGRL method, a random generator based on the existing one in the Longcat application, and a random solution-down generator.

For the evaluation and training, we needed a way to determine the difficulty of Longcat levels, so we also analyzed Longcat levels and, in combination with real user data, determined what aspects of a Longcat level are important for predicting difficulty.

With this approach we formulated the following research questions and found subsequent answers.

1. What attributes of Longcat levels are relevant for predicting difficulty?

   From our analysis phase, we formulated a graph representation for Longcat levels. Variables of these graphs were then used to attempt to predict difficulty. We found that Number of indeterminate branches, Number of solutions, and Number of solution branches were the most useful variables for prediction. Shortest solution length and Number of indeterminate states were also shown to be strong predictors, but they were left unused since they were exploitable by the generator.

2. What percentage of Longcat levels generated with PCGRL and Puzzle-Up are solvable?

   The PCGRL model, based on the turtle method from the original article, was unable to create any solvable levels. The puzzle-up random generator based on the one in the Longcat application generated solvable levels 5.535% of the time. Both these generators use a type of process called puzzle-up, which, from findings, seems to be very ill-suited for generating Longcat levels. The other two generators use a type of process called solution-down that guarantees solvability.

3. What are the differences in difficulty distribution for Longcat levels generated with SD-PCGRL compared to those generated with a random generator?

SD-PCGRL has a more spread-out distribution at a lower non-zero temperature, than the ones generated with the random generators. This means it generates fewer levels of trivial and easy difficulty and more of a challenging difficulty. Its most challenging levels are also more difficult than the random generator's most demanding levels.

4. What percentages of Longcat levels generated with SD-PCGRL are duplicates, and how does it compare with a random generator?

   The puzzle-up random generator gave 68.56% of generated levels as duplicates. The solution-down random generator gave 9.68% of generated levels as duplicates. SD-PCGRL was shown to have a similar number of duplicates as the solution-down random generator at higher temperatures, but the number of duplicates increases as the temperature gets lower. At a temperature of 0.3, which is the best temperature for difficulty distribution, it gave 41.0% of generated levels as duplicates.

5. What percentages of Longcat levels generated with SD-PCGRL have a non-trivial difficulty, and how does it compare with a random generator?

   Of the levels generated with the SD-PCGRL generator, 28.64% of them were solvable, unique, and non-trivial, while only 8.24% were for the solution-down random generator.

From this, we found that the SD-PCGRL was more useful for generating engaging puzzle levels for Longcat than the other generators. We believe this approach of using solution-down generation, difficulty predictors, and reinforcement learning could be used to develop generators for different games. However, this would require a lot of specialized work from a designer, which might be possible for some games. Ultimately, the methods described show promise but must be tested with other games to verify their usefulness.

Bibliography

[1] Maria-Virginia Aponte, Guillaume Levieux, and Stephane Natkin. “Measuring the level of difficulty in single player video games”. In: *Entertainment Computing* 2.4 (2011). Special Section: International Conference on Entertainment Computing and Special Section: Entertainment Interfaces, pp. 205–213. ISSN: 1875-9521. DOI: https://doi.org/10.1016/j.entcom.2011.04.001. URL: https://www.sciencedirect.com/science/article/pii/S1875952111000231.

[2] In-Chang Baek, Sung-Hyun Kim, Seo-yung Lee, Dong-Hyun Lee, and Kyung-Joong Kim. “IPCGRL: Language-Instructed Reinforcement Learning for Procedural Level Generation”. In: *arXiv preprint arXiv:2503.12358* (2025).

[3] Paul Cairns. “Engagement in Digital Games”. In: *Why Engagement Matters: Cross-Disciplinary Perspectives of User Engagement in Digital Media*. Ed. by Heather O’Brien and Paul Cairns. Cham: Springer International Publishing, 2016, pp. 81–104. ISBN: 978-3-319-27446-1. DOI: 10.1007/978-3-319-27446-1_4. URL: https://doi.org/10.1007/978-3-319-27446-1_4.

[4] Barbara De Kegel and Mads Haahr. “Procedural puzzle generation: A survey”. In: *IEEE Transactions on Games* 12.1 (2019), pp. 21–40.

[5] Catherine Doyle, Tara Mertz-Hack, and Joshua Kern. “Do mental activities such as crossword puzzles, playing games, and reading reduce the risk of developing dementia?” In: *Evidence-Based Practice* 20.9 (2017), pp. 13–14.

[6] Sam Earle, Maria Edwards, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. “Learning controllable content generators”. In: *2021 IEEE Conference on Games (CoG)*. IEEE. 2021, pp. 1–9.

[7] Sam Earle, Zehua Jiang, and Julian Togelius. “Scaling, Control and Generalization in Reinforcement Learning Level Generators”. In: *2024 IEEE Conference on Games (CoG)*. IEEE. 2024, pp. 1–8.

[8] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. “Pcgrl: Procedural content generation via reinforcement learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 16. 1. 2020, pp. 95–101.

[9] Raph Koster. *Theory of fun for game design*. " O’Reilly Media, Inc.", 2013.

[10] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N Yannakakis, and Julian Togelius. "Deep learning for procedural content generation". In: *Neural Computing and Applications* 33.1 (2021), pp. 19–37.

[11] Yoshio Murase, Hitoshi Matsubara, and Yuzuru Hiraga. "Automatic making of sokoban problems". In: *PRICAI'96: Topics in Artificial Intelligence: 4th Pacific Rim International Conference on Artificial Intelligence Cairns, Australia, August 26–30, 1996 Proceedings 4*. Springer. 1996, pp. 592–600.

[12] Jonah Segree. *The Case for Procedural Generation in Puzzle Games - Jonah Segree (ThinkyCon 2024)*. Youtube. 2024. URL: `https://www.youtube.com/watch?v=xqmTp20juZk&t=560s`.

[13] Łukasz Spierewka, Rafał Szrajber, and Dominik Szajerman. "Procedural Level Generation with Difficulty Level Estimation for Puzzle Games". In: *International Conference on Computational Science*. Springer. 2021, pp. 106–119.

[14] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. "Procedural content generation via machine learning (PCGML)". In: *IEEE Transactions on Games* 10.3 (2018), pp. 257–270.

[15] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[16] Marc Van Kreveld, Maarten Löffler, and Paul Mutser. "Automated puzzle difficulty estimation". In: *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2015, pp. 415–422.

[17] Yiling Wu, Xiaonan Wang, Skylar Hong, Min Hong, Meng Pei, and Yanjie Su. "The relationship between social short-form videos and youth's well-being: It depends on usage types and content categories." In: *Psychology of Popular Media* 10.4 (2021), p. 467.

[18] Yahia Zakaria, Magda Fayek, and Mayada Hadhoud. "Procedural level generation for Sokoban via deep learning: An experimental study". In: *IEEE Transactions on Games* 15.1 (2022), pp. 108–120.