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

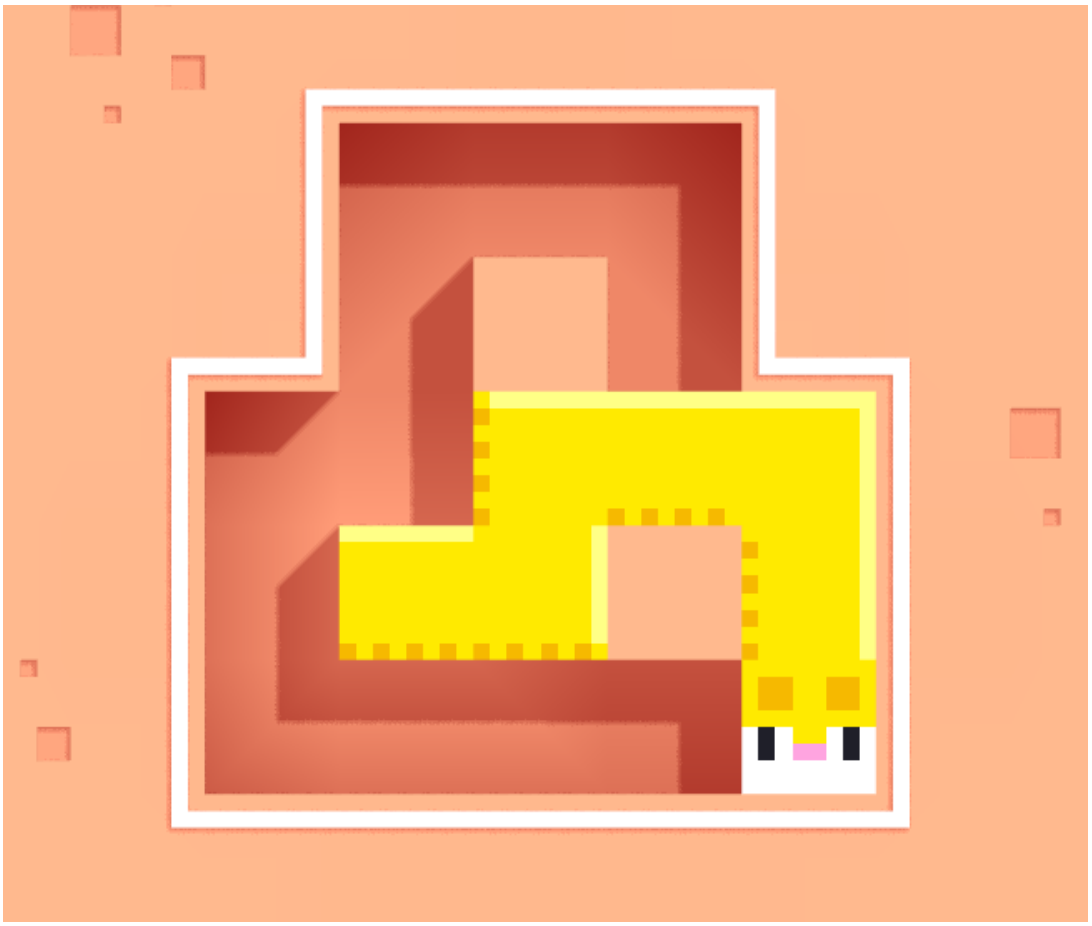

Figure 2.1: Example image from the game Longcat.

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.

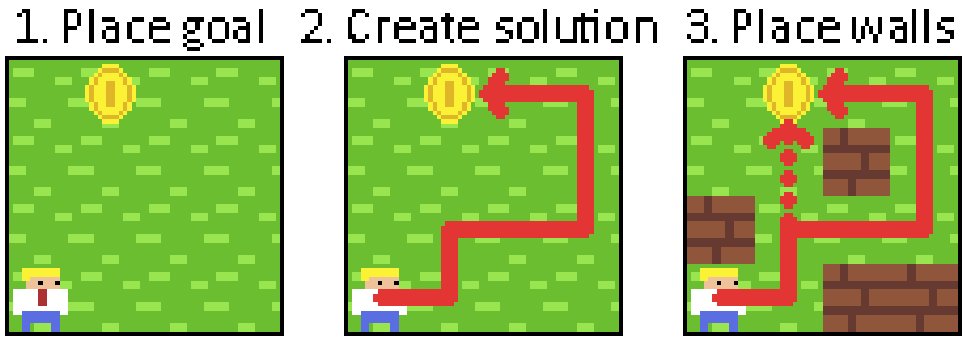

Figure 2.3: Example of a Solution-Down generation method for a maze game.

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.