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.

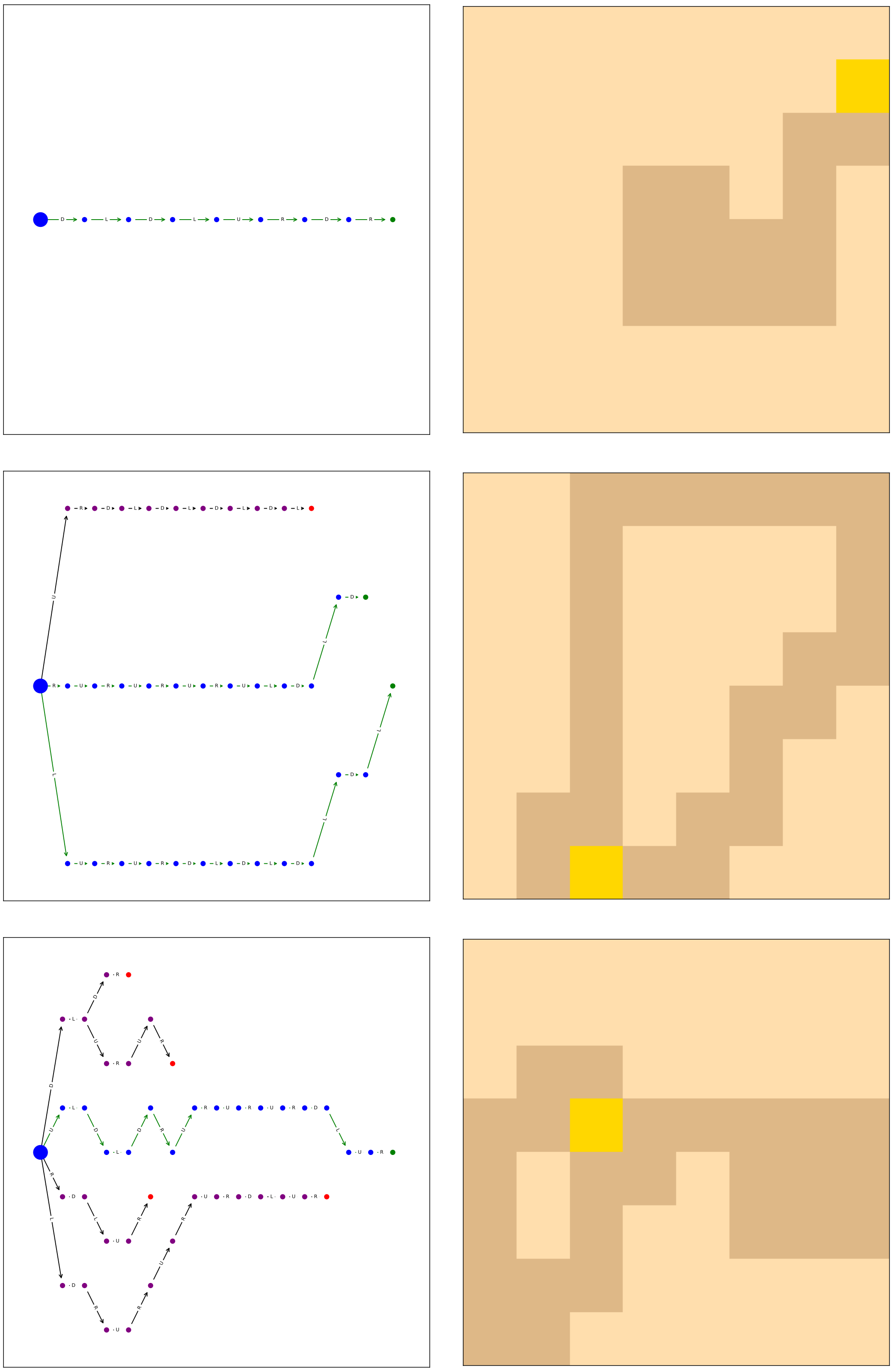

Figure 6.1: Three levels of varying difficulty in the span 0 to 20 sampled from the ones generated by the solution-down random generator.

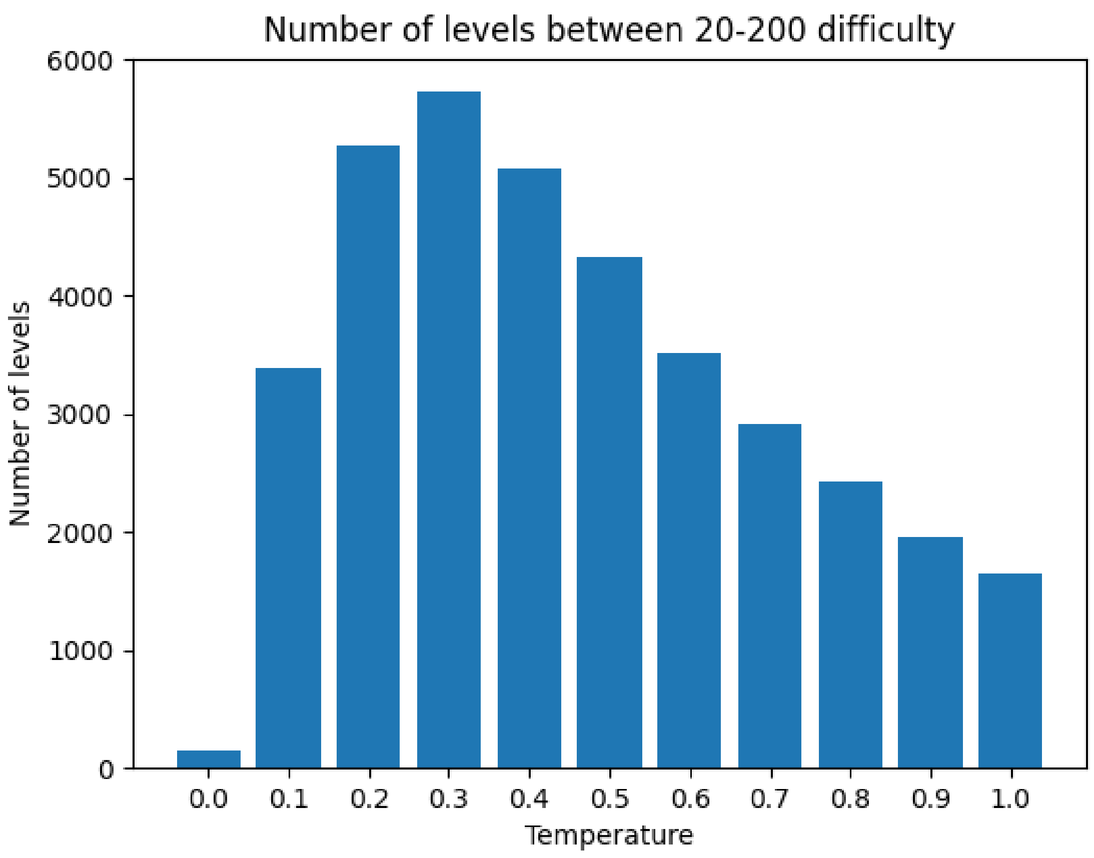

Figure 6.2: Number of non-trivial unique levels generated by the SD-PCGRL generator at each temperature.

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.