Level Assembly as a Markov Decision Process

Colan F. Biemer[1], Seth Cooper[1]

[1]Northeastern University, Boston, MA, USA

Abstract

Many games feature a progression of levels that doesn't adapt to the player. This can be problematic because some players may get stuck if the progression is too difficult, while others may find it boring if the progression is too slow to get to more challenging levels. This can be addressed by building levels based on the player's performance and preferences. In this work, we formulate the problem of generating levels for a player as a Markov Decision Process (MDP) and use adaptive dynamic programming (ADP) to solve the MDP before assembling a level. We tested with two case studies and found that using an ADP outperforms two baselines. Furthermore, we experimented with player proxies and switched them in the middle of play, and we show that a simple modification prior to running ADP results in quick adaptation. By using ADP, which searches the entire MDP, we produce a dynamic progression of levels that adapts to the player.

Keywords

level generation, Markov Decision Process, adaptive dynamic programming

1. Introduction

Many games on the market today feature a form of static level progression (that is, levels are played in a fixed order). An alternative is a level progression that adapts to the player. At face value, such a dynamic approach seems preferable. So, why not use it? First, we need to take into account the workload. A static progression has a minimal number of levels required. A dynamic progression requires multiple levels for different players, requiring more work from already overworked developers [1]. A potential solution is procedural content generation [2], but designers will still need to do additional work [3]. The second problem is adapting to the player. We could use a player model [4, 5] to inform level generation. But the resulting system can be complex and may not have the accuracy guarantees that a designer needs to put the system into production. Overall, creating a dynamic progression has a barrier to entry that is too high for most games.

One of the first cases for adaptive games comes from Hunicke [6], which argues for dynamic difficulty adjustment (DDA). They show how small adjustments and interventions can improve player performance. Jennings-Teats et al. [7] showed a different approach where a generator adapts levels for a player using models trained before play. The problem with models that do not update with the player in real-time is that they may not generalize to all players. Alternatively, online learning is where a model trains in real time. In our opinion, online learning is a more promising approach for dynamic progressions because it addresses the weaknesses of prebuilt models, but there are problems with how quickly these algorithms can adapt to the player.

In our work,[1] we formulate the problem of dynamic progressions as a Markov Decision Process (MDP). (See the Approach section for details on MDPs.) We show how to make an MDP from a directed graph of portions of levels that we append together, where each portion of a level is a state in the MDP. The *director* is the agent that makes decisions about how to assemble levels with the MDP. The rewards of the MDP are initialized via a custom reward function that represents designer preference. The reward table updates based on a dynamic reward function that takes into account designer and player preference as well as how many times the player has played a given state. Before assembling the next level, the director can learn from the updated MDP. In this work, we use policy iteration, which is a method in the family of adaptive dynamic programming (ADP). The benefit of using ADP, which is typically an offline approach, is that we use it in an online environment to effectively learn the best possible level formation before each playthrough.

We test with four directors: Random, Greedy, Policy Iteration (PI), and Adaptive Policy Iteration (API)—a simple modification to PI that we describe below. We apply these directors to two case studies. For the first case study, we initialize an MDP with an n-gram built with *Mario* levels with rewards based on the presence of an enemy in the level slices. An A* agent tests completability and a surrogate player model rewards level slices with at least one enemy. We find that API and PI perform equally well in terms of reward, but API produces levels that are on average more completable. Random is the best performer in terms of completable levels but worst for reward. Our second case study uses an MDP that connects *Kid Icarus*

The Joint Workshop Proceedings of the 2022 Conference on Artificial Intelligence and Interactive Digital Entertainment

biemer.c@northeastern.edu (C. F. Biemer);
se.cooper@northeastern.edu (S. Cooper)

[1]https://github.com/crowdgames/mdp-level-assembly

level segments to form a level. We use player proxies [8, 9] to evaluate level segments, where a proxy has a distribution over the levels it can beat and where it struggles. A proxy also specifies the kind of level segments it enjoys as a player reward. We find that API performs best on average across all the player proxies. Additionally, we test how well directors adapt when a player proxy changes. We find that only API adapts to the change. This lends credence to our argument that ADP with a simple modification, in addition to being straightforward to implement, is an effective method for adapting to players because it explores the whole space, improves data efficiency, and is a promising area for future work in dynamic progressions.

2. Related Work

DDA can be achieved through level generation, and Jennings-Teats et al. [7] gave an early example with *Polymorph*. They used a generator [10] that generated levels based on rhythms (run, jump, or wait). They collected data with this generator, where players played short levels; their playthroughs and results were stored, as well as the player's subjective difficulty ranking. This data was used to train two models to classify difficulty and player skill. By adopting a generate-and-test approach, the game adapted to the player.

One approach to address the limitations of online-learning techniques comes from Gonzalez-Duque et al. [11]. In their work, they used MAP-Elites [12] to generate a grid of levels for multiple agents, where the objective is to find a level with a sixty percent win rate for the target agent. With these grids, they tested how quickly a grid can be updated for a new agent. They achieved this with the intelligent trial-and-error algorithm [13]. In the ideal case, they can adapt to a new agent in one iteration. Overall, this is very similar to our work. We both use the MAP-Elites grid as a starting point and an offline reinforcement learning method in an online environment. However, unlike our work, when working with real players, they need an agent that can reasonably approximate the player for an update, whereas our work does not rely on a surrogate for the player.

One of the earliest formulations of a game as an MDP comes from Thue and Bulitko [14] with the procedural game adaption (PGA) framework. The MDP decides what happens next in the world for a player. Outside of the MDP, there is the *manager*, which estimates the player's reward function and policy. With these two models, the manager updates the MDP. Thus, player decisions affect the MDP, which is optimized in real-time to give the player the best experience. Thue and Bulitko did not directly apply PGA to DDA and instead focused on narrative direction and player preferences (e.g. weapon preference).

Shu et al. [15] built a framework called experience-driven procedural content generation via reinforcement learning, which is composed of four systems: (1) game-playing agent, (2) latent space generator (e.g. a GAN), (3) level repairer, and (4) experience model. This framework is designed after an MDP. A state is a level. An action is a latent vector that is input to a GAN to get a level. The transition model is modeled by a neural network, and it outputs a new state given the previous state as input. The reward is defined by a player experience model.

3. Approach

3.1. Level Assembly as a Markov Decision Process

For a diagram of the system, see Appendix A. Our system starts by *initializing the MDP*. A Markov Decision Process (MDP) is a framework for modeling decision-making for discrete time and state environments. In this work, we use an MDP to build a level assembly policy that is tailored to the player. An MDP is made up of a set of states ($s \in S$), actions ($A(s) \in A$), rewards ($R(s) \in R$), and a transition model ($P(s'|s,a) \in P$). In our case, we represent the problem as a graph where a state is equivalent to a node and an action to an edge. Therefore, the transition model is better represented as $P(s'|s,s_a)$ where s is the starting state, s_a is the target state for the action taken, and s' is the state actually traversed to. A state s represents a portion of a level, be it a single slice or a level segment. We include two additional states: $start$ and $death$. The death state is a terminal node with a reward of -1. All other states have one to many outgoing edges. Each action can result in the player beating and entering the target state or the player losing and entering the death state. We initialize the probability of the player successfully beating the target state to $P(s_a|s,s_a) \leftarrow 0.99$ and $P(death|s,s_a) \leftarrow 0.01$. Note that choosing the death state is not a valid action. In the case studies below, we discuss each MDP in detail, but for now assume the existence of an MDP. In addition to $R(s)$, we have a static rewards table $R_D(s)$, which is the designer rewards table. This table is built by the designer assigning a reward for every level segment s; we use metrics that can be automated, such as counting the number of enemies in a level slice. To initialize $R(s)$ we use $\forall s \in S : R(s) \leftarrow R_D(s)$. In this scheme, the MDP starts with a bias completely toward the designer and, as we show below, updates over time.

The second step is to *build the policy* for level assembly. Given an MDP, a policy $\pi(s)$ chooses a connecting state given the current state as input. Our goal is to find an optimal policy that balances the designer's and the

player's reward—we discuss how a policy is built in the Directors section.

The third step in our system is to *generate a level* with the policy. Imagine we want to build a level three states long. We start with $\pi(start)$ and get s_1. We input s_1 into the policy and get s_2. Lastly, we enter s_2 and get s_3. The assembled level is the concatenation of s_1, s_2, and s_3.

Next, the *player plays the level*. For this example, let's assume that the player completed s_1, failed fifty percent of the way through s_2, and never reached s_3. At this point, the player can *exit* or choose to play again. In the latter case, we *update the MDP* based on their results.

First, we modify the MDP by updating the neighbors of the $start$ node with any level segment the player beat. In this case, the player only beat s_1. If s_1 isn't a neighbor of the start node, the edge $(start, s_1)$ is added—probabilities are handled below. In this case, we know s_1 is already a neighbor of the start node because the policy selected s_1 from the start node. As a result, no edge is added.

$$R(s) \leftarrow \frac{R_D(s) + M(s)}{N(s)} \tag{1}$$

Second, we update $R(s)$ for visited states—s_1 and s_2—with Equation 1. We assume the existence of $M(s)$, which is a player model that quantifies player enjoyment of a given segment after gameplay. $N(s)$ tracks the number of times a state was visited—all state counts are initialized to 1. The value of $N(s)$ is incremented for both s_1 and s_2, but not s_3. By dividing by $N(s)$, we cause reward decay which encourages exploration and disincentivizes repeating states—this may also reduce player fatigue [16].

$$P(s_a|s, s_a) \leftarrow \frac{1 + \sum_{x,s_a \in E} \textit{win}(x, s_a)}{1 + \sum_{x,s_a \in E} \textit{visits}(x, s_a)} \tag{2}$$

Finally, we evaluate the transition probabilities. E is the set of all edges in the graph. We first update the number of times an edge has been used and the number of times the player has beaten the target state. In the example, we update for two edges: $(start, s_1)$ and (s_1, s_2). Visits is incremented for both edges, but only $(start, s_1)$ has the win count incremented. Then we update P with Equation 2 for every edge with the target state of s_1 or s_2. Lastly, we update the probability of entering the death state $P(death|s, s_a) \leftarrow 1 - P(s_a|s, s_a)$ for s_1 and s_2.

After updating the MDP is complete, the system builds a new policy for level assembly.

3.2. Directors

A *director* [17] adjusts the game to ensure the desired player experience. In this work, we use the term to represent an agent responsible for assembling a level.

3.2.1. Random

Random is a baseline director and builds a policy by randomly selecting a neighbor for each state.

3.2.2. Greedy

Greedy is a baseline director and builds a policy by selecting the neighboring state with the highest reward.

3.2.3. Policy Iteration (PI)

The *PI* director uses policy iteration [18], a form of ADP, which approximates the utility of states. For every policy update, the utility of all states are set to 0 and the policy, $\pi_p(s)$, is initialized with random actions. PI then runs *policy evaluation* and *policy improvement* until convergence.[2] Policy evaluation updates the utility of each state using equation 3, and is run k times—we use $k = 20$. γ is a discount factor for future rewards. We used $\gamma = 0.95$, which we found experimentally by running small trials.

$$U_{i+1}(s) \leftarrow \sum_{s'} P(s'|s, \pi(s))[R(s') + \gamma U(s')] \tag{3}$$

Policy improvement updates the policy to π_{i+1} by changing the target state to the state that results in the highest utility according to the updated utility table. If there is a change, the algorithm has not converged and returns to policy improvement. If there is no change, the policy has converged.

3.2.4. Adaptive Policy Iteration (API)

We found in case study 2 that PI struggles to adapt when a player persona changes. This motivated the *API* director. The difference between PI and API is that API keeps a running tally of the player's losing streak and removes edges from the start node based on how long the streak is. To show how this tally is used, consider a scenario where the player lost for the first time. API will remove one edge from $start$ to the neighbor with the largest $R_D(s)$. Say that the player loses again. API will remove two edges. This pattern continues till the player wins (in which case the tally is reset) or $start$ only has one edge (in which case the edge is not removed). By doing so, API attempts to avoid nodes that cause continuous player failure.

[2] We found that convergence for an MDP with $9,453$ states takes < 0.5 seconds running unoptimized Python code.

Director	Reward	Percent Complete
API	$\mathbf{0.1878 \pm 0.0976}$	0.7912 ± 0.3623
PI	$\mathbf{0.1878 \pm 0.0976}$	0.6763 ± 0.4453
Greedy	0.1172 ± 0.0884	$\mathbf{1.0000 \pm 0.0000}$
Random	0.0864 ± 0.0843	0.9798 ± 0.1309

Table 1
Average plus or minus the standard deviation of the reward and percentage completed for each director for Case Study 1 on n-grams for *Mario* level assembly.

3.3. Evaluation

We have two case studies to evaluate our approach. In the first case study, we model n-gram level generation as an MDP. This is a challenging task because n-grams may not have the full context required to make a level that can be completed by an agent. In the second case study, we use previous work [19, 20] that created a graph that connects level segments, and we turn the graph into an MDP. This allows us to always generate completable levels and focus on the player's preference expressed by the player model. Furthermore, the second case study has a much larger MDP in terms of the number of states.

4. Case Study 1: *Mario* N-Grams

4.1. Dataset and Graph

Dahsklog et al. used n-grams to generate full *Mario* levels [21] by breaking a *Mario* level into a set of vertical slices (level slices). We build an n-gram ($n = 3$) with *Mario* levels from the VGLC [22], which are not underground. We use the n-gram to build an MDP where each state represents a prior. There are 513 states with an average of 1.579 actions per state and a max of 47. The designer reward ($R_D(s)$) is 1 if there is an enemy in the state and 0 otherwise.

4.2. Players

Assembling levels with an MDP formed from an n-gram does not guarantee a completable level. We use a modified version of the Summerville A* agent [22] to assess the percent that a level can be completed. As a substitute for a player model ($M(s)$), we define player enjoyment in terms of level slice density (i.e. the number of solid blocks in a column divided by the total number of tiles per column).

4.3. Evaluation

We ran each director twenty times on different seeds with fifty levels generated for each run. Fifty levels was chosen because we view it as a small amount in comparison to *Mario*, which has 32 total levels where players will lose multiple times before beating the game. Each level was thirty level slices long. Sample levels can be seen in Appendix B. The results are in Table 1. In terms of reward, both API and PI are the best performers with an identical average reward and standard deviation. Greedy is the third best performer. Random is the worst. Interestingly, Greedy always produced a completable level. This is the result of getting stuck in a local maxima, which helps explain its low average reward. Next on the list is Random, which performed better in terms of completability than the two ADP methods. API produced levels that were on average more completable than PI. We expected this because API removes potentially problematic states to start from.

5. Case Study 2: *Icarus* Segment Generation

5.1. Dataset and Graph

MAP-Elites [12] uses a grid formed by tessellating the solution space defined by a set of behavioral characteristics (BC) [23]. The grid is made up of cells that contain elites, which are the best solutions found that fit the cell's range of BCs. Gram-Elites [19] is an extension of MAP-Elites with a publicly available dataset[3] of levels for *Kid Icarus*. The BCs used for *Kid Icarus* are density and leniency [23]. The data set also comes with a graph structure to link level segments [20] that connects neighbors with a directed edge if the concatenated result is beatable and does not contain broken in-game structures. If concatenation fails, a linking algorithm attempts to find a *linker* that satisfies the two requirements. If not possible, there is no edge between the two states. If linking succeeds, a state is created between the two that represents the linker found. There are $9,453$ states and an average of 1.877 actions per state with a max of 11. However, there are many linking states and when they are removed we get $1,094$ states with a mean of 8.579 actions per state and a max of 11.

Gram-Elites designed the values for each BC to be between zero and one. For states that correspond to a cell, we assign $R_D(s)$ as the mean of the state's BCs. For linking states, we set $R_D(s)$ as the mean of the designer reward for the two linked states—this makes it fairer for the Greedy director. Note that we assume that the larger R_D is for a state, the more difficult the state. The theory is that the more complex the level, the more difficult the level. This may not be true, and we discuss it more in the Conclusion.

There are two final points. First is reward decay. Gram-Elites has multiple elites per cell, which are very similar

[3] https://github.com/bi3mer/GramElitesData

Player Proxy	Always Wins Threshold	Fail Percent Complete	Player Reward
Bad Player Likes Hard Levels	$D + L < 0.25\ MAX\_BC$	0.25 - 0.40	BC/MAX_BC
Bad Player Likes Easy Levels	$D + L < 0.25\ MAX\_BC$	0.25 - 0.40	$1 - BC/MAX\_BC$
Mediocre Player Likes High Density	$D + L < 0.50\ MAX\_BC$	0.50 - 0.70	D
Mediocre Player Likes High Leniency	$D + L < 0.50\ MAX\_BC$	0.50 - 0.70	L
Good Player Likes Hard Levels	$D + L < 0.75\ MAX\_BC$	0.75 - 0.95	BC/MAX_BC
Good Player Likes Easy Levels	$D + L < 0.75\ MAX\_BC$	0.75 - 0.95	$1 - BC/MAX\_BC$

Table 2
Player proxies used to assess level segments. D stands for density and L for leniency. MAX_BC represents the maximum sum of the BCs, in the case of *Icarus* it is 2. Fail Percent Complete represents the range of percent complete values used if the level is greater than the always win threshold. Player Reward is a substitute for a player model, $M(s)$.

Player	Director	Reward	Percent Complete
Bad Player Likes Easy Levels	API	$\mathbf{0.1698 \pm 0.0767}$	0.6555 ± 0.2986
	PI	0.1522 ± 0.0912	0.5594 ± 0.3368
	Greedy	0.1178 ± 0.0739	0.4046 ± 0.2770
	Random	0.1208 ± 0.0814	$\mathbf{0.9884 \pm 0.0808}$
Good Player Likes Easy Levels	API	$\mathbf{0.5079 \pm 0.2486}$	0.8442 ± 0.2683
	PI	0.4826 ± 0.2519	0.7870 ± 0.3043
	Greedy	0.4965 ± 0.2216	$\mathbf{1.0000 \pm 0.0000}$
	Random	0.5009 ± 0.2143	$\mathbf{1.0000 \pm 0.0000}$
Good Player Likes Hard Levels	API	$\mathbf{0.6093 \pm 0.3150}$	0.9217 ± 0.2053
	PI	0.5751 ± 0.3070	0.8639 ± 0.2716
	Greedy	0.4519 ± 0.2006	$\mathbf{1.0000 \pm 0.0000}$
	Random	0.1323 ± 0.0972	$\mathbf{1.0000 \pm 0.0000}$
Mediocre Player Likes High Density	API	$\mathbf{0.4427 \pm 0.2320}$	0.76840 ± 0.2964
	PI	0.3974 ± 0.2183	0.6278 ± 0.3028
	Greedy	0.4187 ± 0.2140	0.6847 ± 0.3389
	Random	0.1375 ± 0.1205	$\mathbf{1.0000 \pm 0.0000}$
Mediocre Player Likes High Leniency	API	$\mathbf{0.2835 \pm 0.1366}$	0.7089 ± 0.2886
	PI	0.2112 ± 0.1422	0.4494 ± 0.3131
	Greedy	0.1950 ± 0.1015	0.6609 ± 0.3229
	Random	0.1272 ± 0.0858	$\mathbf{1.0000 \pm 0.0000}$
All Players	API	$\mathbf{0.4026 \pm 0.2694}$	0.7797 ± 0.2895
	PI	0.3681 ± 0.2695	0.6269 ± 0.3518
	Greedy	0.3317 ± 0.2285	0.7810 ± 0.3092
	Random	0.2038 ± 0.1972	$\mathbf{0.9977 \pm 0.0364}$

Table 3
Average plus or minus the standard deviation of the reward and percentage completed for each director for Case Study 2 for *Icarus* level assembly.

and may not feel different to the player. We address this by incrementing $N(s)$ for all states in that cell. Second, we gave the example above of generating a level three segments long and counted links as a state. The work on linking level segments guarantees that the resulting level will be completable, but linking states include very few level slices. We address this by not counting them as a full state (e.g. a level supposed to be two states long may have three states where one is a linking state).

5.2. Players

We have the guarantee that the level segments combined together result in a level that can be completed by an agent [19]. Instead of evaluating with an agent, we use player proxies [8, 9]. A player proxy defines the level segments it can beat and how much it enjoyed that level segment. Table 2 shows the player proxies we use.

5.3. Evaluation

5.3.1. Segment Generation

To test segment generation, we ran each director twenty times on different seeds with fifty levels generated for *Icarus* each run. A level is made up of five states; there could be up to four additional linking states. Appendix C shows sample levels. The results are in Table 3. Figure 1 shows heat maps for three players based on the average number of times a player visited a cell; the highest value

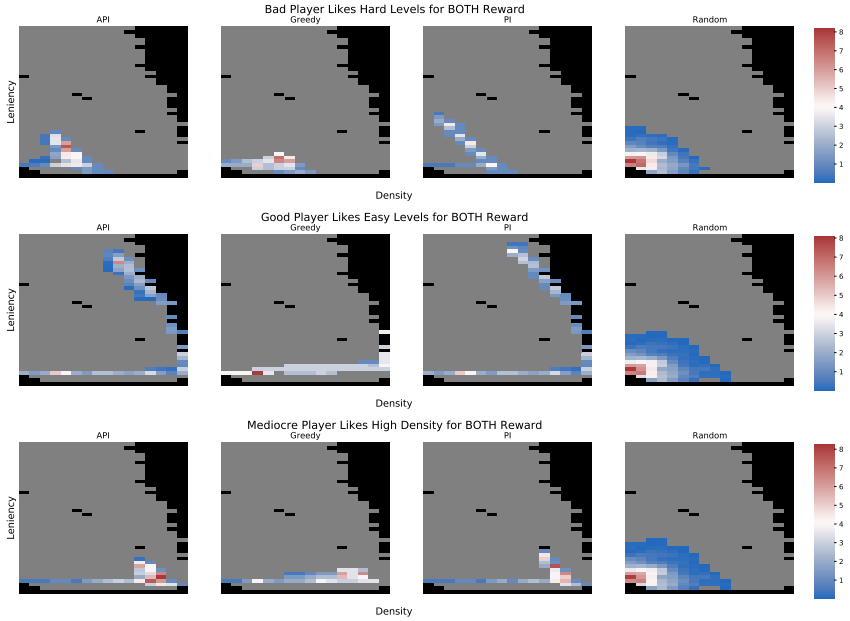


Figure 1: Heat map of cells in the MAP-Elites grid visited by each director for *Icarus* segment generation.

of R_D is at the top right of the heat map.

Starting with the average reward, API is the best performer for all player proxies. PI performs second best for all proxies except *Good Player Likes Easy Levels*, where it is the worst performer. The reason is shown in the middle plot of Figure 1. Both API and PI quickly move to the area with the largest designer reward. Because API reduces failure, it is able to get a larger reward, while PI continually forces the player to try and fail difficult levels. We also see this phenomenon for *Bad Player Likes Hard Levels*, where PI explores the failure front, whereas API finds the front and moves backward. We see this again for *Mediocre Player Likes High Density*.

Moving to percent complete, we can see that the cost for finding levels the player fails is seemingly worse results. However, this is not necessarily problematic. After all, a game where the player never lost would be boring [11]. Random is consistently the best for percent complete across all players since it stays near the origin of the graph. We also see one of the failure points for Greedy on *Good Player Likes Hard Levels* where it is not able to explore the space enough to find a level that is difficult for the player. API and PI, do not have this problem.

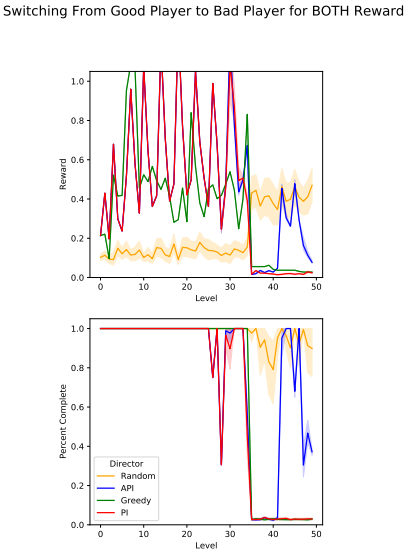


Figure 2: Shows the average reward per level (top) and the average percent complete average per level (bottom) when the player is switched after 35 levels from *Good Player Likes Hard Levels* to *Bad Player Likes Easy Levels*.

5.3.2. Switching Player Proxies

A potential pitfall with systems that adapt to a player is the problem of switching players. A system that fits to a player may only work for that player. When a player switch occurs, we want the director to quickly adapt to the new player. To test this, we ran *Good Player Likes Hard Levels* for 35 levels and then switched to *Bad Player Likes Easy Levels* for 15 levels. These two particular proxies were chosen because the former is an ideal player

proxy given the initial reward function, and the latter is not. When the switch occurs, the MDP will favor harder levels that *Bad Player Likes Easy Levels* cannot beat and does not enjoy. We ran each director twenty times. The directors did not receive notice of the player proxy change. The results are in Figure 2. Initially, API and PI are identical, but there is a slight change when *Good Player Likes Hard Levels* starts to fail. When the player proxy switches, we can see that API, Greedy, and PI get a reward of zero. Random never adapted to the first player and its reward goes up since it happened to stay near the easy levels that the new player liked. From level 35 to 50, we can see that Greedy and PI show no signs of adapting. API adapts after about 7 levels and produces playable levels for the new player proxy.

6. Conclusion

N-gram generation was difficult for every director because using n-grams to form an MDP is an unideal use case. The first problem is that assembled levels are not guaranteed to be completable. Even once a completable level is found, reward decay encourages the director to find a new level, which results in the user experiencing more levels that are not completable. The second problem is that n-gram generation requires that the policy select many level slices to create a single level, which can result in loops of level slices, as seen in Appendix B for API and PI. This could be addressed by updating $N(s)$ and $R(s)$, building a new policy, and then using that policy to select the next level slice during level assembly, but running such a process over and over again would likely be too slow to be worthwhile. Using level segments helps address the problem of looping because the policy is used much less; looping only becomes a problem if the player makes it to the area with the largest reward, in which case the player has ostensibly beat the game.

Our approach is at its best when assembled levels are guaranteed to be completable. API quickly adapts to the player, but could better explore the solution space. A GLIE scheme [24] could address this.

The assumption that the behavioral characteristics we use are a measure of difficulty is unverified. Work has been done that focuses on measuring difficulty [25, 26, 27], and is a promising area area for future work.

It may appear that the reliance on linking for levels formed by level segments is a weakness, but this is not the case. Links create the possibility for more potential levels to be made, but are not required. Further, we can look at *Spelunky*, which has a level generator that mixes handmade segments with procedural design [28]. Instead of a pseudorandom selection of handmade segments, an MDP could be built on top to select segments.

We are interested in two areas for future work. First, we intend to test whether this method is effective as a dynamic difficulty adjustment system with human participants. Second, we want to change our system to use a multi-objective MDP [29, 30] with a finite horizon [31]. The finite horizon will lead to a more accurate utility calculation for the domain of level assembly. Multi-objective optimization reduces the complexity of reward balancing and lets the designer focus more on their game.

References

[1] C. D'Anastasio, Why 2021 was the biggest year for the labor movement in games, 2021. URL: https://www.wired.com/story/2021-biggest-year-labor-movement-video-games/.

[2] N. Shaker, J. Togelius, M. J. Nelson, Procedural content generation in games, Springer, 2016.

[3] I. Karth, A. M. Smith, Addressing the fundamental tension of pcgml with discriminative learning, in: Proceedings of the 14th International Conference on the Foundations of Digital Games, 2019, pp. 1–9.

[4] G. N. Yannakakis, P. Spronck, D. Loiacono, E. André, Player modeling, in: Artificial and Computational Intelligence in Games, 2013.

[5] D. Hooshyar, M. Yousefi, H. Lim, Data-driven approaches to game player modeling: a systematic literature review, ACM Computing Surveys (CSUR) 50 (2018) 1–19.

[6] R. Hunicke, The case for dynamic difficulty adjustment in games, in: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology, 2005, pp. 429–433.

[7] M. Jennings-Teats, G. Smith, N. Wardrip-Fruin, Polymorph: dynamic difficulty adjustment through level generation, in: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, 2010, pp. 1–4.

[8] A. Liapis, C. Holmgård, G. N. Yannakakis, J. Togelius, Procedural personas as critics for dungeon generation, in: European Conference on the Applications of Evolutionary Computation, Springer, 2015, pp. 331–343.

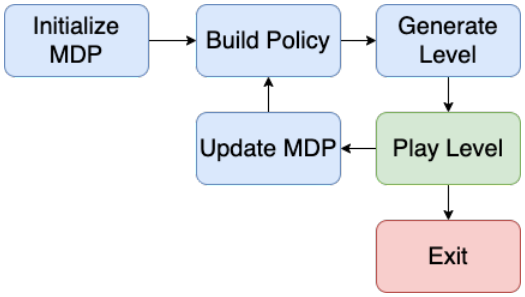
[9] M. C. Green, A. Khalifa, G. A. Barros, A. Nealen, J. Togelius, Generating levels that teach mechanics, in: Proceedings of the 13th International Conference on the Foundations of Digital Games, 2018, pp. 1–8.

[10] G. Smith, M. Treanor, J. Whitehead, M. Mateas, Rhythm-based level generation for 2d platformers, in: Proceedings of the 4th international Conference on Foundations of Digital Games, 2009, pp. 175–182.

[11] M. González-Duque, R. B. Palm, D. Ha, S. Risi, Finding game levels with the right difficulty in a few

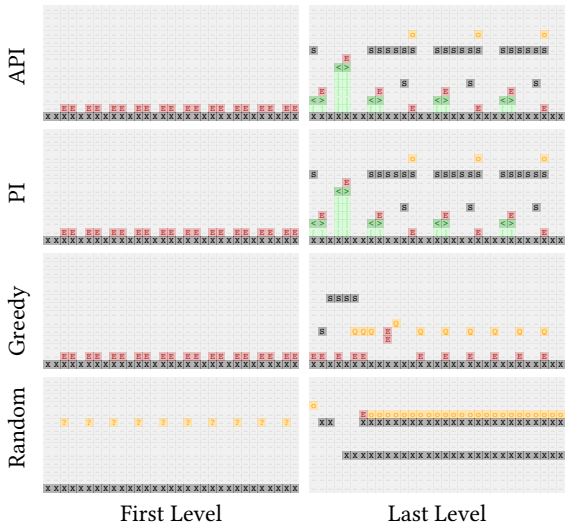
trials through intelligent trial-and-error, in: 2020 IEEE Conference on Games (CoG), IEEE, 2020, pp. 503–510.
[12] J.-B. Mouret, J. Clune, Illuminating search spaces by mapping elites, arXiv preprint arXiv:1504.04909 (2015).
[13] A. Cully, J. Clune, D. Tarapore, J.-B. Mouret, Robots that can adapt like animals, Nature 521 (2015) 503–507.
[14] D. Thue, V. Bulitko, Procedural game adaptation: Framing experience management as changing an MDP, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 8 (2012) 44–50.
[15] T. Shu, J. Liu, G. N. Yannakakis, Experience-driven pcg via reinforcement learning: A super mario bros study, in: 2021 IEEE Conference on Games (CoG), IEEE, 2021, pp. 1–9.
[16] D. Gravina, A. Khalifa, A. Liapis, J. Togelius, G. N. Yannakakis, Procedural content generation through quality diversity, in: 2019 IEEE Conference on Games (CoG), IEEE, 2019, pp. 1–8.
[17] G. N. Yannakakis, Game ai revisited, in: Proceedings of the 9th conference on Computing Frontiers, 2012, pp. 285–292.
[18] R. A. Howard, Dynamic programming and Markov processes, MIT Press, 1960.
[19] C. Biemer, A. Hervella, S. Cooper, Gram-elites: N-gram based quality-diversity search, in: Proceedings of the FDG workshop on Procedural Content Generation, 2021, pp. 1–6.
[20] C. Biemer, S. Cooper, On linking level segments, arXiv preprint arXiv:2203.05057 (2022).
[21] S. Dahlskog, J. Togelius, M. J. Nelson, Linear levels through n-grams, in: Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services, 2014, pp. 200–206.
[22] A. J. Summerville, S. Snodgrass, M. Mateas, S. Ontanón, The vglc: The video game level corpus, arXiv preprint arXiv:1606.07487 (2016).
[23] G. Smith, J. Whitehead, Analyzing the expressive range of a level generator, in: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, 2010, pp. 1–7.
[24] S. Russell, P. Norvig, Artificial intelligence: a modern approach, 3rd edition ed., Pearson, Upper Saddle River, 2009.
[25] M.-V. Aponte, G. Levieux, S. Natkin, Measuring the level of difficulty in single player video games, Entertainment Computing 2 (2011) 205–213.
[26] A. Liapis, G. Yannakakis, J. Togelius, Towards a generic method of evaluating game levels, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2013, pp. 30–36.
[27] J. Fraser, M. Katchabaw, R. E. Mercer, A methodological approach to identifying and quantifying video game difficulty factors, Entertainment Computing 5 (2014) 441–449.
[28] D. Yu, Spelunky: Boss Fight Books# 11, volume 11, Boss Fight Books, 2016.
[29] K. Etessami, M. Kwiatkowska, M. Y. Vardi, M. Yannakakis, Multi-objective model checking of markov decision processes, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2007, pp. 50–65.
[30] F. Delgrange, J.-P. Katoen, T. Quatmann, M. Randour, Simple strategies in multi-objective mdps, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2020, pp. 346–364.
[31] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

A. Appendix A: Diagram of Level Assembly System



B. Appendix B: *Mario* Level Images

The first and last completable level assembled by each director for *Mario* n-grams.



C. Appendix C: *Icarus* Level Images

Last completable level assembled for *Icarus* on the last seed for *Good Player Likes Hard Levels*.