Evaluating the impact of MDP-based level assembly on player experience

Colan Biemer[1], Seth Cooper[1]

[1]*Northeastern University*

Abstract

Dynamic difficulty adjustment adjusts a game to be harder for skilled players and easier for less-skilled players. One way to do this is by changing the level that the player plays, which can be achieved by using a Markov Decision Process (MDP) to assemble smaller segments of levels. However, (1) such an approach has not been tested on real players and (2) there are open questions regarding how the MDP should be configured. In this paper, we evaluate MDP-based level assembly with two games—a platformer and a roguelike—with two player studies, and found that while an automatically generated MDP resulted in a similar player experience, it did not outperform a handcrafted MDP. This shows that MDP-based level assembly can be effective, but more work needs to be done on how to best generate MDPs for level assembly. Additionally, MDP-based level assembly was compared to a static level progression (i.e., the player plays a level until they beat it, and then they can play the next level). We found that the static level progression was too easy for one game and too hard for the other, helping show that a dynamic approach results in a more consistent player experience across players.

Keywords

level assembly, player study, Markov decision process, platformer, roguelike

1. Introduction

The majority of games that do not feature an open world use a static level progression (SLP), which is a level progression where the player plays a level until they beat it, and then they can move on to the next level. They continue to play until the player beats the game or quits. For designers, making an SLP is a challenging task because some players are skilled and others are not. Making an SLP that works for both is nearly impossible, and, as a result, some players will quit because the game is too easy and others because it is too hard [1].

One way to solve this problem is dynamic difficulty adjustment (DDA) [2], which adjusts parts of a game to make it easier or harder based on the player. If they are struggling, a DDA system should make the game easier, and, conversely, it should make the game harder if the game is not challenging enough. Typically, this is achieved via stat adjustments, such as increasing the player's health, increasing damage, and so on [2, 3].

DDA can also be achieved by serving the player different levels [4, 5, 6]. These levels can be selected from a dataset, but they can also be procedurally generated [7] or stitched together from level segments generated offline [8]. However, little work has used the latter approach of assembling levels from level segments in the context of DDA.

To approach this, we built on previous work from Biemer and Cooper [9], which automatically built a Markov Decision Process (MDP) [10] to assemble levels from level segments using a tool called *Ponos*.[1] We extend their work by (1) testing if the approach works with real players by running two player studies with two games (a platformer and a roguelike) with 160 participants for each study, and (2) comparing the output of *Ponos* to an MDP built by hand. As a baseline, we included an SLP condition. This was done to answer two research questions:

12th Experimental Artificial Intelligence in Games Workshop, November 10–11, 2025, Edmonton, Alberta, CA

biemer.c@northeastern.edu (C. Biemer); se.cooper@northeastern.edu (S. Cooper)
http://colan.biemer.us/ (C. Biemer); https://sethcooper.net/ (S. Cooper)
0000-0003-4993-9548 (C. Biemer); 0000-0003-4504-0877 (S. Cooper)

[1] Code for Ponos available on GitHub: https://github.com/bi3mer/ponos.

- **RQ1**: Does a reward based on expected difficulty and enjoyment for the automatically built MDP result in a better player experience than using a simple reward based on depth?
- **RQ2**: Does an automatically built MDP, a handcrafted MDP, or a static level progression result in a better player experience?

For **RQ1**, we found that crafting a complex reward based on expected difficulty and enjoyment was not necessary, when, instead, the structure of the MDP could be used to build a reward; this structure is described in Section 3. For **RQ2**, we found that the SLP for the roguelike was too easy to beat, and the SLP for the platformer was too difficult. In comparison, the MDP-based conditions had better results in terms of how often players won and lost for both games. However, there was little to no impact on the player experience, as defined by the mini Player Experience Inventory (mPXI) [11]. We were able to determine, though, that the MDP-based conditions performed better than the SLP condition for both games.

2. Related Work

Work from Shu et al. [12] showed how PCG can be guided with a player experience model [13]. While they did not test with players, their work is relevant because it raises a question: Instead of using rewards like difficulty or enjoyment to guide level generation, why not use a player experience model? The main reason we didn't use a player experience model was accessibility. The hope is that someone, someday, will use MDP-based level assembly in a commercial game. If the final answer to **RQ2** is that developers also need to build a player experience model, then MDP-based level assembly will likely be out of scope because implementing a player experience model is a significant undertaking. Making it work well for a specific game is even more work. Therefore, a player experience model is not accessible for the majority of developers when the primary engineering focus is on gameplay, performance, and bug fixes.

One valid criticism of using a reward based on difficulty and enjoyment (see **RQ1**) is that neither is concrete; they are concepts, and operationalizing them for a rewards table has many pitfalls. Shouldn't the MDP try to optimize the number of levels played by a player in a session [4] or the likelihood that a player will succeed when playing a level segment [6]? The former provides a real-world example with a Match-3 game in production. They used a progression graph where the reward was based on how many levels it would take to reach a given node from the current node, while also keeping track of the number of trials; a trial is an attempt to beat a level. The reward table had the structure of $R_{k,t}$ where k represented the level and t the number of trials. The more trials failed, the more likely the player was to enter the churn state, which represented the player giving up. An approach like this could be used to create a more dynamic reward table. The approach could be augmented so that the MDP had a new reward table that also considered the number of trials, but this wouldn't be the correct approach because players are not expected to replay the same level repeatedly until they win. Instead, $R_{k,t}$ would have to be converted to R_k, but it is not clear how this could be accomplished.

The reward of difficulty per level segment is subjective. What one player finds difficult, another may find easy. Static player rankings of difficulty are, therefore, inherently flawed, but the range of player disagreement is not clear. For example, a flat level in *Mario* [14] where the player only has to move to the right is "easier" than a flat level where the player has to jump once over a gap. One way to view this is that *difficulty* is the result of in-game challenges [15]. In this formulation, a challenge either impedes progress through a level or ends a playthrough. One way to adjust a level's difficulty is to move challenges in the way of or out of the way of the player's path [16]. This, though, takes us back to a static ranking of difficulty. This same line of reasoning applies to enjoyment, but enjoyment is far more subjective. The hope, therefore, is that the player rankings of difficulty and enjoyment gathered in previous work [17] capture the more objective aspects of difficulty and enjoyment.

This notion of in-game challenges can be linked to required mechanics; for example, the player must know how to jump to get past a gap obstacle. In theory, a game can be broken down into a list of mechanics [18]. A level may have multiple potential paths to beat it, but those paths can also be broken down into a list of required mechanics. A player who has demonstrated knowledge of all the required mechanics will be expected to beat a level with those mechanics. This notion is similar to the work of Butler et al. [19], which built a system for building an automatic game progression. The system's goal was to introduce new mechanics while maintaining an appropriate level of difficulty. The goal of their work is similar to ours, only we do not break down mechanics and instead rely on the behavioral characteristics (BC) [20] of levels while building a progression based on an MDP.

The work from Butler et al. [19] was not a work focused on DDA; it was a work on intelligent tutoring systems (ITS) [21, 22, 23, 24]. The overlap between DDA and ITS has not, to our knowledge, been directly analyzed with a useful visualization, such as a Venn diagram, but the overlap is considerable. The main difference appears to be the goal. DDA aims to keep the player in a state of challenge, while an ITS has the goal of player learning (e.g., learning English grammar [25], how to multiply and divide [26], how to program [27], etc.).

For DDA to work in the medium to long term, players have to learn new mechanics and ways to play the game. One alternative reward that was not explored, but inspired by ITSs was one based on the game's mechanics [28]. Levels with mechanics that the player had not experienced would have higher rewards than levels with mechanics that the player had already demonstrated. Competence could be measured in the number of times the player successfully exhibited a given mechanic. This competence score could be used to penalize levels where the player was unlikely to fail, similar to the work of Butler et al. [19], and a dynamic reward table could be built.

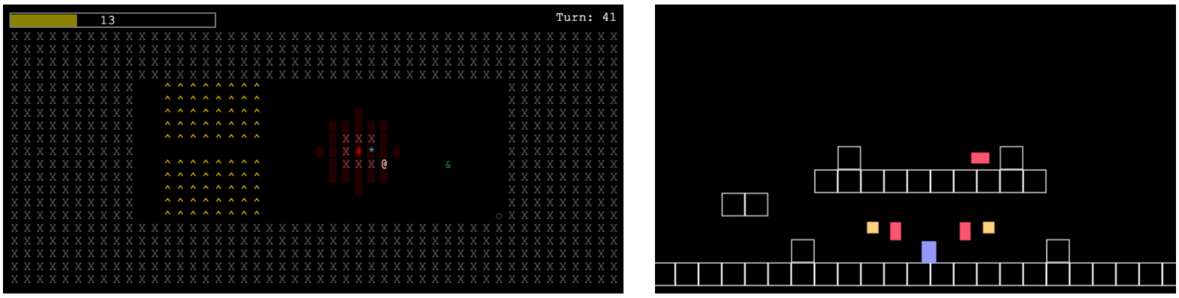
The reason that this mechanics-based reward was not used was, again, accessibility. Proving that a mechanic is required to beat a level is a difficult and computationally expensive problem [29]. One recent solution is to use a constraint solver, which shows that a level cannot be beaten without the given mechanic [30]. Every layer added to the system, however, is a layer of complexity that developers must maintain. A simpler alternative would be to build an approximation of the required mechanics, which looks at level features (i.e., gaps) and elements (i.e., a Goomba in *Mario*). However, the goal for this work is simplicity, but such an approach would be an interesting area for future work.

3. MDP-Based Level Assembly

This paper used the open-source tool *Ponos* to generate MDPs for level assembly. It works by using Gram-Elites [31], an extension of MAP-Elites [32], to generate a set of level segments with n-grams [33, 34] organized by their BCs [20]. A BC is an aspect of a level (e.g., density). The output of Gram-Elites is a k-dimension grid, where k is the number of BCs, and each cell in the grid can contain level segments. This grid is then turned into a digraph based on neighbors, and the edges are validated with a linking algorithm [35] to guarantee completability. The digraph is turned into an MDP [9], where states are level segments and actions are edges from the digraph. The MDP assembles levels by concatenating level segments together that are guaranteed to be completable, as assured by the linking algorithm.

The resulting MDP can be used during a gameplay session. After beating or failing a level, the MDP's reward table is updated based on the player's performance, and a new policy is built for level assembly. For this to work, there are two special states in the MDP: *start* and *end*. *start* is where level assembly begins. Edges from the start node are added when the player beats levels and removed when they lose—see work from Biemer and Cooper [9] for more details, but in this work, we allow the player to lose twice in a row before edges are removed. The *end* state is reached when the player beats the level. For this to work, the *end* state needs a positive reward, and the other states should have a negative reward, because otherwise the MDP will never move towards the *end* state, since the reward horizon would be infinitely positive.

Of special note is the addition to the MDP of a reward table R_D, or the designer reward table. This table, R_D, is how the designer influences the direction of level generation beyond the *end* state and the structure of the grid, as defined by BCs. A main focus of this work is on how to best define R_D. The reward table is initialized to R_D ($\forall s \in S : R(s) \leftarrow R_D(s)$), and future reward updates are influenced by R_D ($R(s) \leftarrow R_D(s)N(s)$, where $N(s)$ is the number of times state the player has visited state s;



(A) *DungeonGrams*

(B) *Recformer*

Figure 1: Example screenshots from *DungeonGrams* and *Recformer*

Biemer and Cooper [9] utilized a surrogate player model in their work, but a player model was not required for the approach to work, and we forgo it in the name of simplicity).

One place where this work differs from the approach offered by Biemer and Cooper [9] was how the transition table $P(s'|s, a)$ was updated. They updated it in a binary fashion based on the number of times a level was beaten divided by the total number of times it was visited. In this work, we changed the numerator to the sum of the percent completed. Meaning, if a player played state s once and completed 90% of that level (i.e., they lost), then the probability of success would be $(1 + 0.9)/(1 + 1)$, (0.95) rather than $(1 + 0)/(1 + 1)$, (0.5). The initialization of success was 99% for all edges based on exploratory playthroughs.

4. Player Study Details

Two games were used in this work. The first was *DungeonGrams* [36] (see Figure 1A). *DungeonGrams* is a roguelike where the player traverses levels to reach a portal, always at the bottom right of each level. The portal does not open until the player has activated every switch in the level. When the player moves, they lose one point of stamina. The player can gain back stamina by collecting food blocks. There can also be enemies in a level. Enemies chase the player when the player is within a certain range of the enemy's starting point. The second game was *Recformer* [37] (see Figure 1B), a platformer where the player has to collect every coin in the level to beat the level. Meaning, unless there is a coin at the far right of the level, the player does not have to go all the way to the right to win. The game includes several enemies that move deterministically (vertically, horizontally, or in an oval). Finally, there are turrets that fire a bullet at the player's location every 2.5 seconds when the player is in range, and lasers that fire upward every 2 seconds when the player is in range.

To gather data on the impact of different MDP level progressions, we ran user studies with both games. Besides the game the participant played, each study was exactly the same. Participants played the game, filled out a survey (demographics, mPXI, and then three custom questions: "The game was too hard," "The game was too easy," and "I felt bored while playing this game") on Qualtrics, and then they were shown a code that they could use to get their payment. 160 players were recruited for each game via Prolific. Study methods had approval from our university's Institutional Review Board.

A server was built to serve static content (the game in the case of this work), store logs, and assign conditions [38] with block randomization [39].[2] The four conditions were *r-mean*, *r-depth*, *static*, and *hand* (conditions are described below). Each condition was assigned 40 participants.

Each participant was paid three dollars for an estimated time of fifteen minutes of participation (estimated pay of $12 an hour). The breakdown of time was a required ten minutes of playing the game, and a maximum expected time of five minutes to fill out the survey. Required time playing the target game was enforced by a timer at the top of the screen, which counted down to zero. Once the timer hit zero, the link to the survey was presented with instructions. After participants completed the survey,

[2] Server code available on GitHub: https://github.com/bi3mer/go-log-study-server.

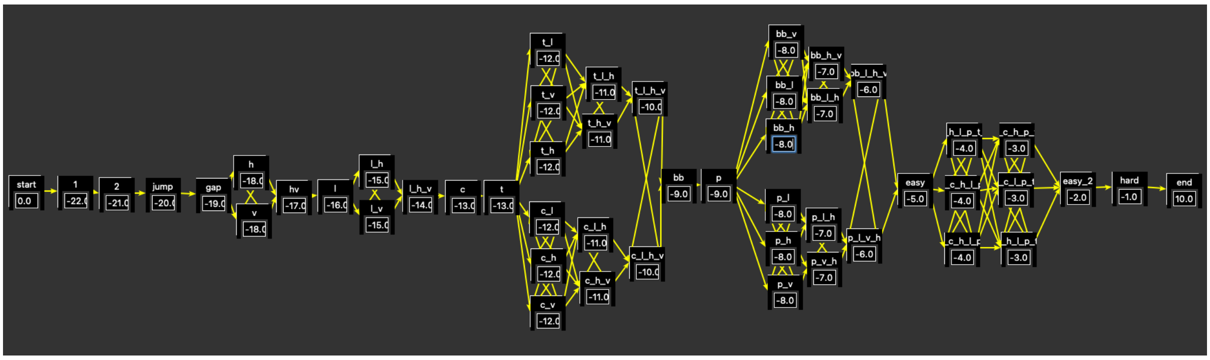


Figure 2: The handcrafted Markov Decision Process built for *Recformer* displayed and built in GDM-Editor, and it is used by the *hand* condition.

they were shown the code to complete their Prolific study. The only way to reach the survey before the ten minutes were up was to beat the game.

Analysis of all Likert data (-3 strongly disagree to 3 strongly agree) was performed with the Kruskal-Wallis test [40]. This was done because ANOVA [41] assumes interval data, and Likert data is ordinal. For any variables that had a p-value < 0.05, a post-hoc analysis was run with Dunn's test [42] with a correction via the Holm method [43] to find any statistically significant differences between groups.

Analysis of behavioral data during gameplay was initially conducted by testing whether ANOVA [41] could be applied with Levene's test [44] to see if the distribution was homoscedastic and the Shapiro-Wilk test [45] to see if the data was approximately normally distributed. For every variable in both studies, the variables examined resulted in at least one of the two tests failing. The Kruskal-Wallis test [40] was run in place of ANOVA, and the same post-hoc methods from before were applied if the p-value was < 0.05.

For *DungeonGrams*, levels were built with 2 level segments because using additional level segments would not guarantee the completability of assembled levels [35]. For *Recformer*, levels were built with 3 level segments, and this was based on experimental playthroughs. Levels made from 2 level segments felt too short to play and 4 too long; note that completability was not an issue for a game like *Recformer* where there weren't long-term dependencies like *stamina* in *DungeonGrams*[35]. Even if 3 level segments had felt too long to play while testing, though, that was the minimum that would have been selected due to a desire to study the player's experience when the MDP director had more influence over the levels assembled.

5. Player Study Conditions

5.1. *hand* Condition

A critical part of any procedural content generation system is *controllability*. Using *Ponos* affords the designer less control over player experience when compared to an SLP. The *hand* condition addresses this by using an MDP built entirely by hand—see Appendix A for details on the process of making an MDP by hand. Figures 2 and 3 show the fully built progressions for *DungeonGrams* and *Recformer*. The MDP built for *Recformer* had 138 nodes and 1161 edges. The path from the *start* node to the *end* node was 26 level segments long. The MDP built for *DungeonGrams* was composed of 104 nodes and 342 edges. The path from the *start* node to the *end* node was 27 level segments long.

5.2. *r-mean* Condition

The *r-mean* condition used an MDP automatically generated by *Ponos*, with a reward which was the average of a level's expected difficulty and enjoyment based on player data from a previous study from Biemer and Cooper [17]. Generation with *Ponos* starts with Gram-Elites [31]. The configurations for

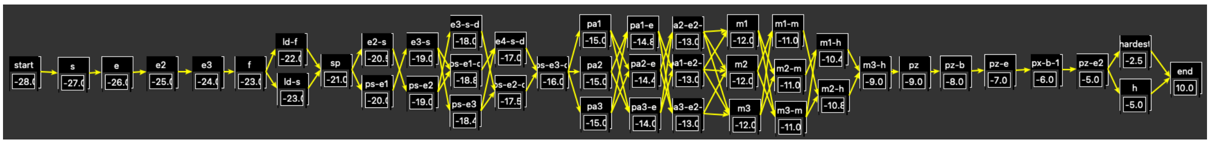


Figure 3: The handcrafted Markov Decision Process built for *DungeonGrams* displayed and built in GDM-Editor, and it is used by the *hand* condition.

both games are in Appendix B. One change made to Gram-Elites was that the configuration no longer required a resolution for each BC. This allowed for binary BCs (e.g., an enemy is in the level) as well as count-based BCs (e.g., number of enemies in a level).

After Gram-Elites is linking [35]. This requires an additional configuration detail of the maximum link length. The maximum link length for *DungeonGrams* was 2 and *Recformer* was 1. Neither game requires structure completion [35], and so that part of the linking algorithm was not used. Once linking was complete, the graph was pruned so every node was reachable from the *start* node and could reach the *end* node.

With the graph complete, the next step for *Ponos* was to build the designer reward table R_D. For *DungeonGrams*, the reward table R_D was built by using the data collected from a previous study [17], which analyzed different heuristics to approximate difficulty and enjoyment of level segments using a linear regression model to predict player rankings of difficulty and enjoyment. We used the output of both models and calculated the mean for the reward. We then transformed it to be between 0 and 1 and subtracted one so that the reward was negative.

However, we did not use the exact same model that Biemer and Cooper found to be best performing for enjoyment. The best model for calculating enjoyment used *path-nothing* (see [17] for a description), but it could not be used for *Recformer*, and thus was not used for either game to better align the two studies. Further, the correlation coefficient for *path-nothing* was -0.017746 and had little impact on the output. The difficulty model for *DungeonGrams* used as input the following computational metrics: *jaccard-nothing*, *proximity-to-enemies*, *stamina-percent-enemies*, and *density* (see [17] for descriptions of these) as input. Some of these, like *jaccard-nothing*, which utilized Jaccard similarity [46] on the paths between two different versions of a level, were grid-based, but were not removed due to being more impactful on calculating difficulty.

The models built for *DungeonGrams* were not used for *Recformer* because the models were built for a roguelike, and *Recformer* is a platformer. Enjoyment for *Recformer* was the output score of *proximity to enemy*—this heuristic looks at the solution path and calculates how close enemies are to the player as they navigate the level—using every frame in the simulation. *path-nothing* was not used for *Recformer* because removing coins would make the level solved. The difficulty score was the mean of *proximity-to-enemies* and a modified version of density. The modification came about because a highly dense level can be just as challenging as a highly sparse level in a platformer. U-curve density was calculated with: $u\_density(lvl) = (solid\_blocks(lvl) - 0.5 * area(level))^2 / (0.5 * area(level))^2$.[3] The mean of the two scores was then calculated, and used to fill in $R_D(s)$.

After the reward was set for every node, it was updated based on the node with the largest reward: $R_D(n) \leftarrow R_D(n)/max(R_D) - 1$. This puts all the rewards in the range of $[-1, 0]$.

The final MDP for *DungeonGrams* had 598 nodes and 2,953 edges. The minimum path from the start node to the end node was 26 nodes long. The final MDP for *Recformer* had 3,362 nodes and 11,575 edges. The minimum path from the start node to the end node was 23 nodes long.

5.3. *r-depth* Condition

The *r-depth* condition used the MDP built by *Ponos* for the *r-mean* condition. The reward, though, was changed to be based on the node's depth, or distance from the *start* node. The update used

[3] u-curve density was not used as a BC for gram-elites because it would allow for highly dense and sparse levels to be in the same gram-elites cell.

the max depth node, which was the end node. Every designer reward was set with: $R_D(n) \leftarrow depth(n)/max(depth) - 1$. Also, note that R_D was static, so when edges were added or removed, R_D wasn't updated.

5.4. ***static*** Condition

The *static* condition did not use an MDP. Instead, it used an SLP. There were three options for how the SLP could be built: (1) build it from scratch, similar to the *hand* condition, (2) use the *hand* condition digraph, and pick a single path through it, or (3) use the MDP built for *r-mean* and *r-depth*, and pick a single path through it.

The first option would be ideal if finding professional game designers and having them build an SLP from scratch were a viable option, but it was not for a multitude of reasons. The alternative would be for one of us, the authors, to create the SLP, but that opens up the results to the criticism that if the SLP condition performed poorly, it was because the designer did a poor job building the SLP—this criticism also applies to the *hand* condition. This same criticism would apply if the second option of building the SLP from the *hand* condition digraph was used. That leaves us with the third option, where the SLP was built from the output of *Ponos*. There are still valid criticisms, but fewer than the two alternatives.

Following the third option, one SLP for *DungeonGrams* and one for *Recformer* was built by running a breadth-first search to find the shortest path from the *start* node to the *end* node, and the output was an array of level segments. The SLP stored an `index` which marked where the player was in the SLP. If the player beat the level, that index was incremented by the number of level segments played. If the player lost, then the index would not be changed, and the player would play the same level again.

The path length for *Recformer* was made up of 23 level segments. Meaning, the player had to beat 8 levels to win. For *DungeonGrams*, the path length was made up of 26 level segments, which resulted in the player having to beat 13 levels to win.

6. Game Study: *DungeonGrams*

For *DungeonGrams*, the median time per participant was 13 minutes and 24 seconds, including time playing the game and filling out the survey. This yielded a median pay of \$13.44 an hour. Two participants played the game but did not complete the survey, and five participants completed the survey without playing a level. All seven were dropped from the dataset. This left 39 participants for the *hand* condition, 39 participants for the *r-depth* condition, 39 participants for the *r-mean* condition, and 37 participants for the *static* condition. The median age range for *hand* was 25-34, and the median age range of the other three conditions was 35-44. No participants beat the game for *hand*, *r-depth*, and *r-mean*. However, 26 of the participants assigned to the *static* condition beat the game.

Table 1 shows the results from the mPXI [11] and the three custom survey items with a compact letter display to distinct groupings found with post-hoc analysis. Only two items had statistically significant results, and neither was on the mPXI; we could not find a difference in player experiences across all conditions. The two that had statistically significant differences dealt with (1) the game was too easy to play, and (2) the game was too hard to play. The results for the first ("Too Easy") had a p-value of less than 0.05, but the post-hoc tests showed no major difference between the conditions.

Participants mostly disagreed or slightly disagreed with the statement, "The game was too hard to play." Three groups were found in post-hoc analysis. The first contained *hand* and *r-depth*, the second *r-depth* and *r-mean*, and the third *r-depth* and *static*. The comparisons show that players disagreed less that *hand* was too hard as compared to *r-depth* and *static*, and disagreed more that *static* was too hard as compared to *hand* and *r-mean*. One interpretation of this is that *hand* was harder than *static*, with the other two falling somewhere in the middle.

Table 2 shows the behavioral results from playing the game. For "Time per Player," you may have expected to see 600 seconds (10 minutes) for every condition. The ten-minute time limit began when the webpage was opened; however, the metric does not count time spent on the main menu, the tutorial, or the two-second transition scene (win/loss screen). In other words, "Time per Player" shows how long

	hand	r-depth	r-mean	static
Audiovisual Appeal ($p = 0.821$)	2.0 (1.0)	2.0 (1.0)	2.0 (0.0)	2.0 (1.0)
Challenge ($p = 0.869$)	2.0 (1.0)	2.0 (1.0)	1.0 (1.0)	2.0 (1.0)
Ease of Control ($p = 0.243$)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)
Clarity of Goals ($p = 0.804$)	2.0 (0.0)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)
Progress Feedback ($p = 0.860$)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)
Autonomy ($p = 0.759$)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)
Curiousity ($p = 0.620$)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)
Immersion ($p = 0.365$)	3.0 (0.0)	3.0 (0.0)	3.0 (0.0)	3.0 (0.0)
Mastery ($p = 0.140$)	1.0 (1.0)	2.0 (1.0)	1.0 (1.0)	2.0 (1.0)
Meaning ($p = 0.818$)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)
Too Easy ($p = 0.046$)	0.0 (1.0)^{a}	0.0 (2.0)^{a}	-1.0 (1.0)^{a}	**1.0** (1.0)^{a}
Too Hard ($p < 0.001$)	**-1.0** (1.0)^{a}	-2.0 (1.0)^{bc}	-2.0 (1.0)^{ab}	-2.0 (1.0)^{c}
Bored ($p = 0.283$)	-2.0 (1.0)	-2.0 (1.0)	-2.0 (1.0)	-2.0 (1.0)

Table 1
Results for all survey questions related to Likert data for participants who completed at least one level for *DungeonGrams*. The format for each condition is "[[median]] ([[median absolute deviation]])". For any survey question with a p-value less than 0.05, the title and the largest median is bolded, and groupings are displayed with a compact letter display.

	hand	r-depth	r-mean	static
Time per Player ($p < 0.001$)	409.029 (84.274)^{a}	403.488 (82.967)^{a}	**416.632** (51.284)^{a}	298.423 (91.323)^{b}
Time per Level ($p < 0.001$)*	15.221 (12.858)^{a}	17.923 (13.410)^{b}	17.286 (14.778)^{c}	**18.842** (15.594)^{d}
Levels Played ($p < 0.001$)	**26.872** (10.641)^{a}	22.513 (10.655)^{a}	24.103 (12.140)^{a}	15.838 (3.908)^{b}
Levels Won ($p = 0.046$)	11.128 (4.450)^{a}	9.641 (4.252)^{a}	10.000 (4.397)^{a}	**11.973** (1.966)^{a}
Levels Lost ($p < 0.001$)	**15.744** (6.724)^{a}	12.872 (7.021)^{a}	14.103 (8.070)^{a}	3.865 (3.857)^{b}
Lost by Enemy ($p < 0.001$)	**11.154** (5.731)^{a}	8.103 (5.462)^{a}	9.436 (7.016)^{a}	2.459 (3.342)^{b}
Lost by Stamina ($p < 0.001$)*	3.744 (3.053)^{a}	**4.154** (2.975)^{a}	4.000 (4.051)^{a}	0.622 (1.714)^{b}
Lost by Spike ($p = 0.694$)*	0.846 (0.863)	0.615 (0.702)	0.667 (0.613)	0.784 (0.904)

Table 2
Quantitative results from the player study for *DungeonGrams*. Results are in the format of "[[mean]] ([[standard deviation]])". Each category with a '*' superscript failed to pass the Shapiro-Wilk test and/or Levene's test, and the Kruskal-Wallis test was used instead, else ANOVA was used. For any variable with a p-value less than 0.05, the title and the largest mean is bolded, and groupings are displayed with a compact letter display.

the player played *DungeonGrams* with their assigned condition. *static* had about 100 seconds less time played than the other three conditions. This was because *static* was the only condition where players beat the game, and those players did not play the full ten minutes. Responses for "Challenge" and "Too Easy" did not reflect so many *static* players winning. "Too Hard," though, did reflect this result, but only minimally, and recall that *static* was grouped with *r-depth*.

"Time per Level" was different for each condition. Players, on average, spent the most time per level for the *static* condition and the least amount of time on the *hand* condition. The result was unexpected because it was thought that players would spend less time on average playing levels in the *static* condition, as they would have to spend less time analyzing the level the next time they played if they lost. However, instead, the effect appears to have been that players in the dynamic conditions kept playing while players in the *static* condition took more time for each level. Alternatively, more time was spent per level because players in the *static* condition lost less.

"Levels Played," "Levels Lost", "Lost by Enemy," and "Lost by Stamina" all displayed a statistically significant difference with the dynamic conditions being in one group and *static* in another. These differences can be attributed to the number of levels played, with fewer levels played by the *static* condition players who beat the game.

	hand	r-depth	r-mean	static
Audiovisual Appeal ($p = 0.626$)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)
Challenge ($p = 0.130$)	2.0 (0.0)	1.5 (0.5)	2.0 (1.0)	1.0 (1.0)
Ease of Control ($p < 0.001$)	**2.0** (0.5)^{a}	1.5 (0.5)^{b}	**2.0** (1.0)^{ab}	**2.0** (1.0)^{b}
Clarity of Goals ($p = 0.952$)	2.5 (0.5)	2.0 (1.0)	2.0 (1.0)	2.5 (0.5)
Progress Feedback ($p = 0.368$)	2.0 (1.0)	2.0 (1.0)	1.0 (1.0)	1.0 (1.0)
Autonomy ($p = 0.678$)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)	1.5 (1.5)
Curiousity ($p = 0.360$)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)	2.0 (1.0)
Immersion ($p = 0.547$)	3.0 (0.0)	3.0 (0.0)	3.0 (0.0)	3.0 (0.0)
Mastery ($p = 0.088$)	2.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.5)
Meaning ($p = 0.300$)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.5)
Too Easy ($p = 0.015$)	**0.5** (0.5)^{a}	0.0 (1.0)^{ab}	-1.0 (1.0)^{ab}	-1.0 (1.0)^{b}
Too Hard ($p = 0.101$)	-2.0 (1.0)	-1.0 (1.0)	-1.0 (1.0)	0.0 (1.5)
Bored ($p = 0.386$)	-2.0 (0.5)	-3.0 (0.0)	-2.0 (1.0)	-2.0 (1.0)

Table 3
Results for all survey questions related to Likert data for participants who completed at least one level for *Recformer*. The format for each condition is "[[median]] ([[median absolute deviation]])". For any survey question with a p-value less than 0.05, the title and the largest median is bolded, and groupings are displayed with a compact letter display.

7. Game Study: *Recformer*

The median time per participant was 14 minutes and 40 seconds, including time spent playing the game and filling out the survey, yielding a median pay of $\$12.27$ an hour. The *r-depth* condition was re-run separately from the initial study due to an error in the condition's reward assignment, and no participants from the initial study were included in the re-run. Eleven participants played the game but did not complete the survey, and one participant completed the survey without playing a level. All twelve were dropped from the dataset. This left 36 participants for the *hand* condition, 38 participants for the *r-depth* condition, 34 participants for the *r-mean* condition, and 38 participants for the *static* condition. The median age range for *static* was 25-34, and the median age range of the other three conditions was 35-44. Five of the participants beat the game for the *static* condition, and one participant beat the game for the *hand* condition; no player beat the game for the remaining conditions.

Table 3 shows the results for the Likert questions for *Recformer*. For the majority, the null hypothesis could not be rejected. Only "Ease of Control" and "Too Easy" had p-values that were less than 0.05.

"Ease of Control" was represented by the statement: "It was easy to know how to perform actions in the game." It had two groups. In the first was *hand* and *r-mean*. In the second was *r-depth*, *r-mean*, and *static*. The first group had slightly higher scores for ease of control, closer to "Agree." There was a statistically significant difference between *static* and *hand* as well as *r-depth* and *hand*.

There were two groups for "Too Easy." The first group was composed of *hand*, *r-depth*, and *r-mean*. The second was composed of *r-depth*, *r-mean*, and *static*. The difference, then, was between *hand* and *static*, with participants assigned the *hand* condition between neutral and slightly agreeing that *Recformer* was too easy, and participants assigned the *static* condition slightly disagreeing.

Table 4 shows the behavioral results for *Recformer*. Unlike in *DungeonGrams* where there was a two-second transition between levels, *Recformer* had a win/loss screen where the player had to press the space button to start the next level. As a result, the time of less than 600 seconds (the ten-minute time limit) indicates how quickly players were pressing the spacebar to play the next level and the amount of time they spent on the main menu before playing.

The first row in Table 4 where the null hypothesis could be rejected was for "Time per Level." There were two groupings. The first was for all the dynamic conditions, and the other was for the *static* condition. The dynamic conditions had a higher time per level, at around 17 seconds played per level; recall that this result is the opposite of what we found for *DungeonGrams* where players spent more time on levels when playing the *static* condition.

	hand	r-depth	r-mean	static
Time per Player ($p = 0.371$)*	483.545 (49.641)	460.104 (65.847)	453.951 (101.094)	458.730 (63.945)
Time per Level ($p < 0.001$)*	**17.530** (17.236)^{a}	17.277 (16.256)^{a}	17.401 (16.407)^{a}	15.905 (16.221)^{b}
Levels Played ($p = 0.696$)	27.583 (12.451)	26.632 (7.842)	26.088 (10.288)	28.842 (10.559)
Levels Won ($p < 0.001$)	**13.694** (8.246)^{a}	7.632 (2.776)^{b}	8.765 (4.577)^{b}	3.737 (2.061)^{c}
Levels Lost ($p < 0.001$)	13.889 (6.653)^{a}	19.000 (6.031)^{b}	17.324 (7.198)^{ab}	**25.105** (10.995)^{c}
Death by Fall ($p < 0.001$)	3.139 (3.137)^{a}	7.447 (3.306)^{b}	5.294 (4.335)^{ab}	**11.105** (7.976)^{c}
Death by Enemy ($p = 0.208$)*	10.750 (7.041)	11.553 (6.248)	12.029 (5.838)	14.000 (7.951)

Table 4
Quantitative results from the player study for *Recformer.* Results are in the format of "[[mean]] ([[standard deviation]])". Each category with a '*' superscript failed to pass the Shapiro-Wilk test and/or Levene's test, and the Kruskal-Wallis test was used instead, else ANOVA was used. For any variable with a p-value less than 0.05, the title and the largest mean is bolded, and groupings are displayed with a compact letter display.

The next row with a statistically significant difference was "Levels Won," which had three groups. The first was made of *r-depth* and *r-mean*. The second contained only *hand*. The third and final group only had the *static* condition. *hand* had the most levels won on average, *r-depth* and *r-mean* were in the middle, and the *static* condition had an average of only 3.692 levels beaten per participant. The majority of participants in the *static* condition got stuck and could not progress.[4] And recall here that "Challenge" from the mPXI showed no statistically significant differences among the conditions, and that participants in the *static* condition only slightly disagreed with the statement, "The game was too easy to play."

For "Levels Lost," there were three groups. The first group contained *r-depth* and *r-mean*. The second had *hand* and *r-mean*. The final group had the *static* condition. *static* had the most levels lost by a wide margin. *r-depth* and *r-mean* had less levels lost on average, and *hand* had the least. Recall that "Levels Played" had no statistically significant differences, so the difference can be better attributed to the conditions. Interestingly, although *static* participants lost more than the other conditions, there was no negative effect on the player's experience.

Finally, the last category with a p-value less than 0.05 was "Death by Fall." *static* featured the most deaths by falling (see the previous footnote for a short description of the third and fourth level in the progression). *r-depth* and *r-mean* were in the second group with less falls than *static*. The third group contained *r-mean* and *hand*, with the least amount of deaths by falling.

8. Discussion

For **RQ1**, there were minor differences between *r-depth* and *r-mean*, but there was only one statistically significant difference between the two across both user studies, and that was for "Time per Level" in *DungeonGrams*. The answer, then, is that something simple like depth is enough to result in a positive player experience given a good structure for the digraph built by *Ponos*. However, that is not to say that more work could not go into defining different reward functions that could have a larger effect on the player experience.

For **RQ2**, both games achieved fairly high scores across all categories of the mPXI. This could be related to recruiting participants from a crowdsourcing platform, where participants were happy to play any game instead of performing other tasks, such as labeling images. This could help explain why, despite the apparent difficulty of the *static* condition in *Recformer* (where most participants only beat 3-4 levels), the median participant only slightly agreed that *Recformer* was neither too easy nor too hard.

For *DungeonGrams*, there were no statistically significant differences in the results for the mPXI. For *Recformer*, only "Ease of Control" was different among the conditions, and the difference was minor,

[4] The third and fourth level in the static condition feature simple jumps from high platforms to low platforms, with one complicated one where the player has to avoid a vertical enemy. Overall, neither level should be challenging for an experienced player. However, the static condition does not adapt to player failure, so the challenge was too early in the game.

with both groups found to be near the "Agree" response. However, there is more to a player's experience than what they report.

One such factor was whether the player beat the game. For *DungeonGrams*, 26 players beat the game, and they were all in the *static* condition. Based on this, the SLP for *DungeonGrams* was too easy. For *Recformer*, five players managed to beat the *static* condition, but the majority could not make it past the third and fourth levels. These results taken together show that the experience of playing the *static* condition was highly variable. If something was wrong, and there was for both games, nothing could be done while the player was playing to alleviate the problem.

Of course, a weakness in the argument is that the SLP was generated automatically. Perhaps if the *static* condition had been built by a designer for both games, the results would have been better balanced. Or, maybe the SLP should have been generated from the *hand* condition instead. Nevertheless, recall that *r-depth* and *r-mean* both used the same digraph as the one used to generate the SLP for *static*. The levels that were too easy and too hard were also possible to serve to the player, and the objective experience of too many wins or losses could have occurred, but that didn't happen. For *DungeonGrams*, players won almost as many levels as the *static* condition, but also lost just as many because they were challenged earlier and more often. For *Recformer*, the participants clearly struggled with playing a platformer, but were still able to get almost double the wins and make progress.

With the *static* condition no longer in consideration, this leaves us with the three MDP-based conditions. There is a difference for *DungeonGrams* on the survey item "Too Hard" between *hand* and *r-depth*, with *r-depth* disagreeing that the game was too hard and *hand* slightly disagreeing. Otherwise, the automatically generated conditions resulted in comparable player experiences to the handcrafted progressions.

Because the *DungeonGrams* study did not yield any major differences, this leaves us with the *Recformer* study. From the results, it is clear that the digraph built by *Ponos* was flawed. One way to see that there was a problem is via the win rates. The win rate for *hand* was 49.65%, whereas the win rate for *r-depth* was 28.66% and for *r-mean* was 33.60%. While it is not clear what the ideal win rate is, the win rates for both *r-mean* and *r-depth* are low when compared to the *DungeonGrams* study, where the win rate for *hand* was 41.41%, for *r-depth* was 42.82%, and for *r-mean* was 41.49%. However, what the ideal win-rate is is something only the designer can answer. For example, the win-rate for *Recformer* would be high for a game like *Super Meat Boy* [47].

We can not definitively answer **RQ2** since the player experiences, as defined by the mPXI, were statistically similar across all eleven categories. However, we can say that the MDP-based approach outperformed SLPs.

9. Conclusion

The player experience was similar across all four conditions for both player studies. This result was a surprising one. For example, one would expect that the *static* condition for *DungeonGrams*, which had a win rate of 75.60%, would have a lower Challenge score than conditions with a lower win rate—the other three had win rates around 40%—but this was not the case. Instead, the scores appeared to be more based on the experience of playing the game rather than in terms of success and failure.

Despite this, we were able to partly answer both research questions. For **RQ1**, we found that a simple reward based on depth resulted in a similar experience to a more complex reward that used difficulty and enjoyment. For **RQ2**, the *static* condition was the worst-performing: players beat most levels and almost never lost in *DungeonGrams*, but made almost no progress while playing *Recformer*. With static removed, that left the three MDP-based conditions, but no further conclusions could be made regarding which was best. As part of future work, we plan to further investigate **RQ2** using new games to better understand the impact of MDP-based level assembly on player experience.

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly to: Grammar and spelling check, Paraphrase and reword. After use, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Czikszentmihalyi, Flow: The psychology of optimal experience, New York: Harper & Row, 1990.

[2] 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.

[3] 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.

[4] S. Xue, M. Wu, J. Kolen, N. Aghdaie, K. A. Zaman, Dynamic difficulty adjustment for maximized engagement in digital games, in: Proceedings of the 26th International Conference on World Wide Web Companion, 2017, pp. 465–471.

[5] 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.

[6] M. Gonzalez-Duque, R. B. Palm, S. Risi, Fast game content adaptation through bayesian-based player modelling, in: 2021 IEEE Conference on Games (CoG), IEEE, 2021, pp. 01–08.

[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] M. C. Green, L. Mugrai, A. Khalifa, J. Togelius, Mario level generation from mechanics using scene stitching, arXiv:2002.02992 [cs] (2020).

[9] C. F. Biemer, S. Cooper, Level assembly as a markov decision process, in: Proceedings of the Experimental AI in Games Workshop, 2023.

[10] S. Russell, P. Norvig, Artificial intelligence: a modern approach, 3rd edition ed., Pearson, Upper Saddle River, 2009.

[11] A. Haider, C. Harteveld, D. Johnson, M. V. Birk, R. L. Mandryk, M. Seif El-Nasr, L. E. Nacke, K. Gerling, V. Vanden Abeele, minipxi: Development and validation of an eleven-item measure of the player experience inventory, Proceedings of the ACM on Human-Computer Interaction 6 (2022) 1–26.

[12] 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.

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

[14] Nintendo, Super Mario Bros., 1985.

[15] M.-V. Aponte, G. Levieux, S. Natkin, Measuring the level of difficulty in single player video games, Entertainment Computing 2 (2011) 205–213.

[16] G. Berseth, M. B. Haworth, M. Kapadia, P. Faloutsos, Characterizing and optimizing game level difficulty, in: Proceedings of the 7th International Conference on Motion in Games, 2014, pp. 153–160.

[17] C. Biemer, S. Cooper, Solution path heuristics for predicting difficulty and enjoyment ratings of roguelike level segments, in: Proceedings of the 19th International Conference on the Foundations of Digital Games, 2024, pp. 1–8.

[18] 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.

[19] E. Butler, E. Andersen, A. M. Smith, S. Gulwani, Z. Popović, Automatic game progression design through analysis of solution features, in: Proceedings of the 33rd annual ACM conference on human factors in computing systems, 2015, pp. 2407–2416.

[20] 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.

[21] A. T. Corbett, K. R. Koedinger, J. R. Anderson, Intelligent tutoring systems, in: Handbook of human-computer interaction, Elsevier, 1997, pp. 849–874.

[22] A. C. Graesser, M. W. Conley, A. Olney, Intelligent tutoring systems. (2012).

[23] D. S. McNamara, G. T. Jackson, A. Graesser, Intelligent tutoring and games (itag), in: Gaming for classroom-based learning: Digital role playing as a motivator of study, IGI Global, 2010, pp. 44–65.

[24] H. S. Nwana, Intelligent tutoring systems: an overview, Artificial Intelligence Review 4 (1990) 251–277.

[25] M. I. Alhabbash, A. O. Mahdi, S. S. A. Naser, An intelligent tutoring system for teaching grammar english tenses, European Academic Research 4 (2016) 1–15.

[26] S.-C. Shih, C.-C. Chang, B.-C. Kuo, Y.-H. Huang, Mathematics intelligent tutoring system for learning multiplication and division of fractions based on diagnostic teaching, Education and Information Technologies 28 (2023) 9189–9210.

[27] C. J. Butz, S. Hua, R. B. Maguire, A web-based intelligent tutoring system for computer programming, in: IEEE/WIC/ACM International Conference on Web Intelligence (WI'04), IEEE, 2004, pp. 159–165.

[28] M. C. Green, L. Mugrai, A. Khalifa, J. Togelius, Mario level generation from mechanics using scene stitching, in: 2020 IEEE Conference on Games (CoG), IEEE, 2020, pp. 49–56.

[29] A. M. Smith, E. Butler, Z. Popovic, Quantifying over play: Constraining undesirable solutions in puzzle design., in: FDG, 2013, pp. 221–228.

[30] S. Cooper, M. Bazzaz, Literally unplayable: On constraint-based generation of uncompletable levels, in: Proceedings of the 19th International Conference on the Foundations of Digital Games, 2024, pp. 1–8.

[31] C. Biemer, A. Hervella, S. Cooper, Gram-Elites: N-gram based quality-diversity search, in: Proceedings of the 16th International Conference on the Foundations of Digital Games, FDG '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–6. doi:10.1145/3472538.3472599.

[32] J.-B. Mouret, J. Clune, Illuminating search spaces by mapping elites, arXiv preprint arXiv:1504.04909 (2015).

[33] 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 &amp; Services, 2014, pp. 200–206.

[34] D. Jurafsky, Speech & language processing, Pearson Education India, 2000.

[35] C. F. Biemer, S. Cooper, On linking level segments, in: 2022 IEEE Conference on Games (CoG), 2022, pp. 199–205.

[36] S. Cooper, C. F. Biemer, Dungeongrams, 2025. URL: https://github.com/crowdgames/dungeongrams.

[37] C. F. Biemer, Recformer, 2025. URL: https://github.com/bi3mer/recformer.

[38] M. Kang, B. G. Ragan, J.-H. Park, Issues in outcomes research: an overview of randomization techniques for clinical trials, Journal of athletic training 43 (2008) 215–221.

[39] J. Efird, Blocked randomization with randomly selected block sizes, International journal of environmental research and public health 8 (2011) 15–20.

[40] P. E. McKight, J. Najab, Kruskal-wallis test, The corsini encyclopedia of psychology (2010) 1–1.

[41] L. St, S. Wold, et al., Analysis of variance (ANOVA), Chemometrics and intelligent laboratory systems 6 (1989) 259–272.

[42] A. Dinno, Nonparametric pairwise multiple comparisons in independent groups using dunn's test,

The Stata Journal 15 (2015) 292–300.

[43] S. Holm, A simple sequentially rejective multiple test procedure, Scandinavian journal of statistics (1979) 65–70.

[44] J. L. Gastwirth, Y. R. Gel, W. Miao, The impact of levene's test of equality of variances on statistical theory and practice (2009).

[45] S. S. Shapiro, M. B. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (1965) 591–611.

[46] P. Jaccard, Nouvelles recherches sur la distribution florale, Bull. Soc. Vaud. Sci. Nat. 44 (1908) 223–270.

[47] Team Meat, Super Meat Boy, 2010.

[48] B. Cowan, B. Kapralos, A simplified level editor, in: 2011 IEEE International Games Innovation Conference (IGIC), IEEE, 2011, pp. 52–54.

[49] M. Guzdial, N. Liao, J. Chen, S.-Y. Chen, S. Shah, V. Shah, J. Reno, G. Smith, M. O. Riedl, Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators, in: Proceedings of the 2019 CHI conference on human factors in computing systems, 2019, pp. 1–13.

[50] A. J. Summerville, S. Snodgrass, M. Mateas, S. Ontanón, The vglc: The video game level corpus, arXiv preprint arXiv:1606.07487 (2016).

A. Making a Handcrafted MDP

The process to build an MDP by hand began by building level segments. Levels are typically built with a level editor [48, 49]. For level segments in this work, we expected a grid-based layout in two dimensions. This means that all that is required is a simple text file with rows of characters, where each character represents a different entity in the game, similar to the Video Game Level Corpus [50]. A text editor (i.e., vi, vim, or nvim) was more than enough.

The next step was to connect the level segments into a digraph, where each level segment was a node. Editing a digraph by hand with a file format like JSON was possible, but it became error-prone as the size of the graph increased. To handle this problem, a graph editor was built called GDM-Editor.[5] GDM stands for graph-based decision making, and is a tool that was built for making graph-based MDPs.[6] See Figures 2 and 3 for examples of the graph editor built for this work. The editor includes basic functionalities like adding and removing edges, updating rewards, and reading from a directory for when the designer makes new level segments.

One problem found while using the editor was that it became cumbersome to use if the designer wanted to create multiple-level segments of a similar type to connect to another group of level segments of a similar type. For example, if the goal was to have five level segments connect to five other level segments, you would have to make 25 edges. By allowing a node to have multiple-level segments, this problem was solved. However, this comes with the sacrifice that each level segment was assigned the same reward. In terms of the final representation, the multi-level node decomposes into k individual nodes with the same reward. Further, the designer has to do extra work to make sure that all 25 possible edges are valid edges in terms of completeness.

The process of making level segment progressions by hand for each game was iterative, and the primary goal was to introduce a concept and then ramp up the challenge. Then, this was repeated until every concept had been introduced. To make sure that the MDP did not result in a linear progression, like the SLPs, concepts (e.g., a horizontal enemy in *Recformer* and a movement pattern in *DungeonGrams*) were introduced in pairs or trios where the player could learn about one, and then learn about another with the already learned concept in the next level segment. This, though, led to some repetitive patterns and made it so players who reached the end level segments had challenging level segments that made up a whole level, resulting in unideal pacing. Therefore, some "easy" level segments were placed in

[5] Code Available on GitHub: https://github.com/bi3mer/GDM-Editor

[6] Code Available on GitHub: https://github.com/bi3mer/GDM

between particularly challenging level segments that served as a break, but also as a jumping off point for players because they were likely to beat the easy level segments.

B. Gram-Elites Configurations

- *DungeonGrams*
  - *N-gram*: All input levels were broken into vertical level slices, or columns. The n-gram was a tri-gram. The input levels were the level segments created for the handcrafted MDP and those from the previous work, totaling 164 input levels.
  - *Generated segment size*: 12 columns.
  - *N-Gram Generation Iterations*: $1,000$.
  - *MAP-Elite Iterations*: $50,000$.
  - *Padding*: A padding of two empty columns was added to either side of the level. The left side padding added the player to the top-left corner of the level. A portal was added to the bottom right corner of the level.
  - *BCs*:
    * Density: returns $min(1, solid\_block\_count/(0.75 * area))$. (Resolution: 20)
    * Enemies: returns $enemy\_count * 8$.
    * Switches: returns $switch\_count * 4$.[7]
- *Recformer*
  - *N-gram*: All input levels were broken into vertical level slices, or columns. The n-gram was a tri-gram. The input levels are the level segments made for the handcrafted MDP. Input levels were the 138 levels made for the *hand* condition.
  - *Generated segment size*: 15 columns.
  - *N-Gram Generation Iterations*: $1,000$.
  - *MAP-Elite Iterations*: $10,000$.
  - *Padding*: A padding of two columns was added to the left and right. The columns were empty, besides a solid block at the bottom. The far right column had one coin.
  - *BCs*:
    * Inverse Density: returns $1 - density(level)$. (resolution: 10)
    * Vertical enemy count: Returns $vertical\_enemy\_count$.
    * Horizontal enemy count: Returns $horizontal\_enemy\_count$.
    * Circle enemy count: Returns $circle\_enemy\_count$.
    * Laser count: Returns $laser\_count$.
    * Turret count: Returns $turret\_count$.
    * Coin count: Returns $coin\_count$.
    * Blue block count: Returns $blue\_block\_count$.

[7] The exact numbers of multiplying by 8 for enemies and 4 for switches were found experimentally, but they also reflect the final resolution used for density. The idea here is to say that the number of enemies and, to a lesser degree, the number of switches, is more important than the exact density of the level.